MASTER EN SCIENCES INFORMATIQUES

Utilisation de la technique du Type-based homeomorphic embedding dans la spécialisation on-line d'interpréteurs

Pierre, Olivier

Award date:
2008

Awarding institution:
Universite de Namur

Utilisation de la technique du *Type-based Homeomorphic Embedding* dans la spécialisation on-line d'interpréteurs

Olivier Pierre

3 septembre 2008

Résumé

L'idée de la spécialisation est d'adapter un programme à des entrées particulières pour rendre le programme plus performant par rapport à ces entrées. Cette technique peut être utilisée dans de nombreux domaines, mais dans le cadre de ce travail, nous allons nous limiter à un certain type de spécialisation, celui de la spécialisation de programmes logiques (appelé déduction partielle) effectuée de manière on-line. Dans ce cas, un domaine particulier rencontre encore des difficultés, c'est celui de la spécialisation on-line de méta-interpréteurs. On va d'abord présenter les différentes connaissances nécessaires pour comprendre ce problème. On va ensuite essayer d'adapter une découverte récente dans les techniques de terminaison de la déduction partielle, à savoir le *Type-Based Homeomorphic Embedding*, au cas particulier de la spécialisation de méta-interpréteurs. Les résultats trouvés lors de ce travail montrent des améliorations effectuées dans la spécialisation de méta-interpréteurs, mais également des limites qu'il faudra dépasser pour mener à bien la spécialisation on-line de n'importe quel méta-interpréteur.

Abstract

The idea of specialisation is to adapt a program with specific inputs to make this program more efficient according to these inputs. This technology may be use in different areas, but in this work, we will only work with one kind of specialisation, the on-line specialisation of logic programs (called on-line partial deduction). In this particular case, a branch has still difficulties ; it's the branch of meta-interpreters's specialisation. We will first introduce the basic knowledge requires to understand this problem. Then, we will try to adapt a new breakthrough in termination of partial deduction méthods (the *Type-Based Homeomorphic Embedding*), to the particular case of meta-interpreters specialisation. The results of this work show some improvements in the field of the meta-interpreters specialisation, but also limits that we will need to surpass to be able to perform the specialisation of any meta-interpreter.

Remerciements

Je tiens à remercier particulièrement Wim Vanhoof, qui a supervisé tout au long de l'année la réalisation de ce travail. Je remercie également François Degrave, Julien Libert et Nicolas Pierre pour leur aide lors de la correction.

Table des matières

Chapitre 1

Introduction

La spécialisation de programme est un domaine assez récent de l'informatique. Brièvement, l'idée de la spécialisation est d'adapter un programme à des entrées particulières, pour rendre le programme plus performant par rapport à ces entrées. Cette technique peut être utilisée dans de nombreux domaines, comme dans la phase d'optimisation de la compilation, dans l'amélioration de programmes lourds, comme les applications graphiques [7], les réseaux neuronaux [11], la gestion des opérations aériennes [2],... et finalement, dans la spécialisation de programme déclaratifs (et plus précisément, de programmes logiques). C'est cette dernière qui sera traitée dans ce mémoire.

Dans le cas de la spécialisation de programmes logiques, un domaine particulier rencontre encore des difficultés : c'est celui de la spécialisation de méta-interpréteurs. Mais récemment, des progrès ont été réalisés dans un autre domaine de la spécialisation logique et, à première vue, il se pourrait que ces avancées puissent être utilisées également dans le cas de notre problème de spécialisation de méta-interpréteurs. Le but de ce mémoire est de vérifier cette hypothèse.

Comme il s'agit d'une partie de l'informatique bien spécifique et peu connue, la première tâche du mémoire est de présenter les connaissances nécessaires pour appréhender correctement le problème de la spécialisation, puis celui de la déduction partielle qui est la technique de spécialisation utilisée dans ce cas particulier.

Les premiers chapitres sont donc consacrés à une synthèse des avancées dans cette branche et du détail des connaissances requises pour bien débuter la partie plus concrète de ce travail.

Tout d'abord, il sera question d'une courte présentation de la spécialisation de programme et des différentes applications de cette technique. Ensuite, nous présenterons les bases de la programmation logique, car c'est ce paradigme de programmation que nous utiliserons tout au long de ce travail. Nous appliquerons alors le principe de spécialisation à des programmes logiques, et plus particulièrement au cas qui nous intéresse, celui de la spécialisation de méta-interpréteurs. Dans ce domaine particulier, nous présenterons où se situe

la recherche actuelle et quels sont les principaux obstacles qui en ralentissent l'avancée.

Ensuite viendra la présentation de la nouvelle méthode proposée pour résoudre certains problèmes liés à la spécialisation. Nous tenterons alors d'appliquer cette technique au problème encore ouvert de la spécialisation de méta-interpréteurs. Plusieurs exemples seront présentés pour illustrer l'utilisation de cette nouvelle technique. Au final, nous serons en mesure de tirer des conclusions sur son efficacité.

Chapitre 2

La spécialisation de programme

2.1 Idée de base

L'idée de base de la spécialisation de programme est de transformer un programme en *fixant* certaines des ses variables, puis d'effectuer certaines opérations qui ne dépendent que de ces variables fixées, dans le but de créer un programme équivalent [1] mais techniquement plus efficace (c'est-à-dire dont l'exécution sera plus rapide, utilisera moins de ressources, ...).

Imaginons une fonction simple dont le but est d'élever un nombre à une puissance donnée :

puissance(x,exp) = if(exp=1) then x else (x puissance(x,exp-1))*

Supposons que dans un programme donné P, on n'aura seulement besoin de mettre un élément au cube, et donc, on n'utilisera que la fonction *puissance(x,3)* (la variable *exp* étant une constante valant 3 dans ce programme). Dans ce cas, la fonction puissance n'est plus optimale pour P, on peut dès lors la *spécialiser* à ce programme pour augmenter les performances. Une façon de spécialiser cette fonction automatiquement serait d'exécuter les parties du programme qui ne dépendent que des entrées connues. Dans ce cas-ci, cela donnera :

*puissance(x,3)= x*puissance(x,2) = x*x*puissance(x,1) = x*x*x*

D'où, on obtiendra un nouvelle fonction qui pourra être utilisée à la place des appels *puissance(x,3)*. On peut représenter cette fonction de la façon suivante :

*puissance'(x) = x*x*x*

1. Deux programmes sont équivalents s'ils donnent un résultat identique pour une entrée identique, quel que soit l'entrée.

Un autre exemple de spécialisation serait de spécialiser un programme par rapport à certaines de ses entrées. Dans le cas où une valeur revient assez souvent en argument du programme, il est parfois intéressant de spécialiser ce programme par rapport à cette entrée particulière. Contrairement à l'exemple au-dessus, on ne va pas *fixer* des valeurs constantes déjà codées dans le programme, mais bien les entrées de ce programme, qui peuvent évidemment varier à chaque exécution.

Prenons comme exemple un programme simple P (écrit dans un langage abstrait), où la variable *entrée* représente une valeur passée en argument au programme, et où la fonction *display* affiche une variable :

```
x = entrée + 2
display(x)
```

Si, pour une raison où une autre, on réalise que la valeur de *entrée* est très souvent 2, on peut alors créer un programme P' adapté à cette entrée. En utilisant le même principe quand dans le premier exemple cela donnera :

```
display(4)
```

Ce programme sera utilisé à la place du programme initial, à chaque fois que l'argument donné pour *entrée* sera de 2, selon le principe de la figure 2.1.

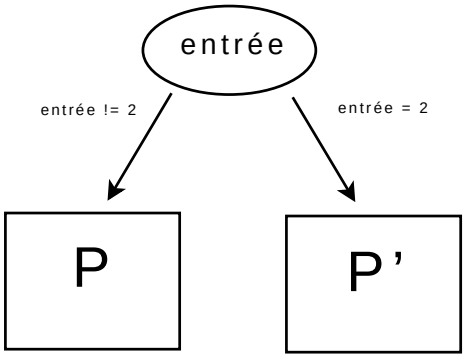

FIGURE 2.1 – Spécialisation selon les entrées

On peut, de la même manière, créer autant de programmes spécialisés qu'il existe d'entrées possibles. Mais une tele méthode pourrait devenir contre performante étant donné l'explosion de la mémoire nécessaire pour stocker un programme aussi simple. De plus, il ne faut pas oublier que la spécialsation par rapport à certaines entrées doit finalement être rentable. Si on spécialise pour des entrées qui ne seront utilisées qu'une seule fois, on perdra du temps. Mais si au contraire on a spécialisé des entrées revenant régulièrement, alors sur plusieurs exécutions on aura regagné le temps perdu par la phase de spécialisation. Pour qu'une spécialisation soit effectivement efficace pour n exécutions, il faut que l'équation suivante soit vérifiée :

$$\begin{gathered} \textit{Temps de spécialisation} + (\ \textit{nouveau temps d'exécution} * n\ ) \\ \leq \\ (\ \textit{ancien temps d'exécution} * n\ ) \end{gathered}$$

On constate ici que la spécialisation est une technique à double tranchant, parce qu'en voulant faire mieux, on risque de faire pire.

Si on le souhaite, on peut également spécialiser un programme à chaque utilisation de ce dernier. Dans ce cas, la spécialisation se fait sur les entrées que reçoit le programme dans un cas particulier. La technique est la même qu'expliqué ci-dessus, si ce n'est qu'une phase de spécialisation est nécessaire avant chaque utilisation de l'application.

Comme le but de la spécialisation reste l'amélioration des performances, il ne faut pas que le temps de spécialisation additionné au temps que met le nouveau programme pour terminer, ne dépasse le temps que l'ancien programme mettrait pour terminer :

$$\begin{gathered} \textit{Temps de spécialisation} + \textit{nouveau temps de terminaison} \\ \leq \\ \textit{ancien temps de terminaison} \end{gathered}$$

Nous dirons dans ce qui suit, que les variables instanciées au moment de la spécialisation sont les variables dites *statiques* et que celles qui seront encore présentes au moment de l'exécution du programme sont appelées *dynamiques*.

De plus, on parlera de spécialisateur ou de fonction de spécialisation pour désigner une fonction qui prend comme argument un programme et ses entrées, et donne comme résultat un programme équivalent à celui en entrée, dans lequel il ne reste plus que les variables dynamiques.

2.2 Applications diverses de la spécialisation

De nombreux domaines profitent des avancées techniques offertes par la spécialisation. Par exemple, la spécialisation de programmes lourds, comme les applications graphiques [7], les réseaux neuronaux [11], la gestion des opération aériennes [2],... Il existe aussi des puces électroniques utilisant certains des aspects de la spécialisation [6]. Deux autres applications intéressantes sont détaillées dans la suite : l'optimisation de compilateurs et la spécialisation d'un point de vue théorique.

2.2.1 Optimisation de compilateurs

Le but d'un compilateur est de transformer un programme d'un langage quelconque en un programme équivalent écrit dans un autre langage, souvent un langage de plus bas niveau. Un bon compilateur va, naturellement, essayer d'optimiser les performances des programmes qu'il transforme. Pour cela, il

peut utiliser des techniques issues de la spécialisation. On peut donner comme exemple la méthode dite de propagation de constante, où, de façon similaire à l'exemple plus haut, les variables statiques sont remplacées par une constante. Cela résultera en un programme compilé plus performant, car on aura retiré des références inutiles vers une variable qui ne changera pas de valeur. D'autres techniques sont également utilisées, on peut notamment citer : l'optimisation des boucles, le retrait de branches if-then-else inutiles,...

Bien sûr, dans le cas de l'amélioration de compilateurs, la spécialisation n'est pas utilisée au maximum de ses capacités, car on ne peut pas spécialiser en fonction d'entrées particulières. La spécialisation n'est effectuée qu'une fois, on perd donc un des avantages principaux de cette technique.

2.2.2 Utilité théorique

L'idée même de spécialisation a permis d'avancer d'un point de vue théorique, principalement dans la théorie de la calculabilité. En effet, la spécialisation associée à la notion de compilation et d'interprétation, permet des résultats intéressants.

Prenons un programme P , les entrées statiques s et les entrées dynamiques d, ajoutons que [] représente le résultat d'une exécution sur le programme entre crochets. Selon la théorie, exécuter un programme avec ses entrées (statiques et dynamiques) est équivalent à exécuter le spécialisateur sur le programme et ses entrées statiques, puis à exécuter le résultat sur les entrées dynamiques :

$$[P](s,d) = [[spec](P,s)]d$$

Il est également évident que l'exécution d'un interpréteur auquel on donne le programme P et l'entrée e est équivalent à l'exécution du programme P prenant l'entrée e.

$$[int](P,e) = [P]e$$

On sait aussi qu'un compilateur prenant un programme en argument va permettre de créer un programme équivalent mais dans un autre langage. Ce qui donne :

$$[comp](P) = P' \ tel \ que \ [P'] = [P]$$

A partir de ces trois formules, on peut déduire ce qu'on appelle les projections de Futamura. La première affirme que l'on peut obtenir une *compilation* en utilisant un spécialisateur sur un interpréteur :

$$[[comp]P]e = [P]e = [int](P,e) = [[spec](int,P)]e$$

d'où :

$$[comp]P = [spec](int,P)$$

La seconde énonce que l'on peut obtenir un compilateur également à partir d'un spécialisateur et d'un interpréteur. Ce qui peut être vérifié en partant de la première projection :

$$[[comp]P] = [[spec](int, P)] = [[[spec](spec, int)]P]$$

d'où :

$$comp = [spec](spec, int)$$

On apprend donc que l'existence d'un spécialisateur et d'un interpréteur pour un programme P suffit à prouver l'existence d'une compilateur pour un tel programme.

2.3 Spécialisation off-line contre spécialisation on-line

Jusqu'ici nous avons expliqué les utilisations possibles de la spécialisation de programme. Voyons maintenant des méthodes plus concrètes afin d'illustrer les applications pratiques d'une telle technique.

La spécialisation, que nous appelerons aussi *évaluation partielle* (car il s'agit de la technique de spécialisation utilisée dans ce mémoire), peut être décrite comme un mélange d'évaluation et de génération de code. Le problème de base des spécialisateurs est de décider *quelles* parties d'un programme vont être spécialisées. Il faut tenir compte de deux facteurs pour choisir correctement :

- Le processus de spécialisation doit toujours se terminer (pas de boucle infinie).
- Le gain en performances du programme spécialisé par rapport au programme initial doit être optimal.

Si on reprend le premier exemple, mais où cette fois le premier argument sera fixe et où l'exposant restera variable, on aura :

puissance(3,exp)

Où il n'y a pas de parties de code qui dépendent uniquement de l'entrée statique. En tentant une spécialisation simpliste on peut supposer :

puissance(3,exp) = if(exp=1) then 3 else (3 puissance(3,exp-1))= ...*

Si on spécialise de cette façon, *puissance(3,exp)* ne se termine pas naturellement. Il faut donc des procédés pour s'assurer que l'on termine toujours la spécialisation et pour qu'elle soit le plus efficace possible. Ces procédés sont séparés en deux grandes familles que l'on appelle *on-line* et *off-line*, ils sont décrits dans la suite.

2.3.1 Spécialisation on-line

Un programme spécialisé de façon on-line est évalué partiellement par un système qui contrôle de façon continue le processus de spécialisation. C'est ce système qui peut décider à tout moment d'arrêter l'évaluation, de générer du code, de reprendre la spécialisation à un autre endroit. C'est un système où tout se passe *en direct*, le spécialisateur recevant le programme et les entrées partielles génère un nouveau programme adapté en conséquence.

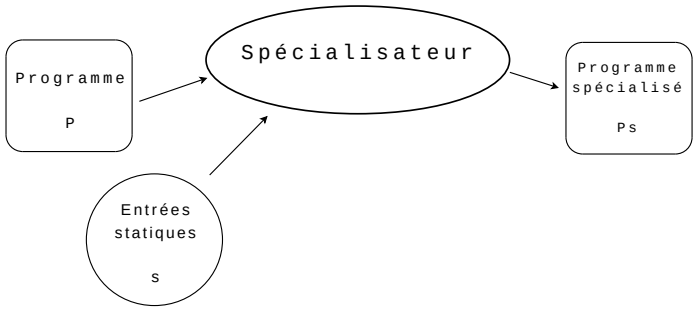

FIGURE 2.2 – Spécialisateur on-line

Les spécialisateurs on-line gardent en général en mémoire tout l'historique de l'évaluation, et utilisent ces informations pour prendre des décisions. Étant donné que le spécialisateur retient tout ce processus d'évaluation, cela en fait un outil puissant qui, en général, donne une meilleure spécialisation qu'un spécialisateur off-line. Par contre, du côté du temps de spécialisation il est perdant, car c'est un procédé assez lourd que de devoir contrôler en permanence le processus d'évaluation. La spécialisation on-line est donc souvent plus lente.

2.3.2 Spécialisation off-line

En ce qui concerne la spécialisation off-line, celle-ci fonctionne en deux phases. La première est appelée la phase de *Binding-Time Analysis*[2]. Cette phase ne tient pas compte concrètement des entrées partielles d'un programme, mais se base plutôt sur une *description* de ces entrées. Elle doit ensuite faire la différence entre les entrées qui seront utilisées pendant la phase de spécialisation (les variables statiques) et celles qui serviront uniquement pendant l'exécution du programme (les variables dynamiques). À partir de ces informations (et seulement ces informations, car on ne dispose pas ici des vraies valeurs que vont avoir les variables ni de l'historique de la spécialisation déjà effectuée) l'analyse va choisir les parties à spécialiser dans la deuxième phase. Le résultat de la *Binding-Time Analysis* est souvent le programme d'entrée auquel est ajouté un commentaire pour chaque action, décrivant si elle doit être évaluée ou pas. Par exemple, reprenons le programme puissance et annotons-le pour qu'un

2. *Analyse, lors de la spécialisation, du moment où les variables pourraient être remplacées par une valeur physique.*

spécialisateur off-line puissent voir facilement quelles parties peuvent être spécialisées (celles qui sont soulignées), ce qui permettra au spécialisateur d'évaluer correctement le programme.

puissance(x, exp) =
if(exp=1) then x else (x puissance(x,exp-1))*

La deuxième phase est la spécialisation proprement dite, elle se fait sur base du nouveau programme annoté, et elle est bien plus simple que pour la spécialisation on-line, puisqu'il suffit au spécialisateur de suivre les commentaires dans le programme pour savoir ce qu'il doit spécialiser. Cette spécialisation est donc plus rapide, mais également moins efficace puisque pendant la phase d'analyse on ne disposait que des descriptions des variables d'entrée. Le schéma 2.3 représente les différentes phases d'une spécialisation off-line.

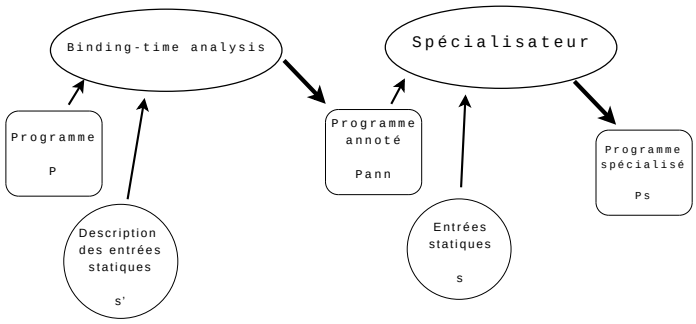

FIGURE 2.3 – Spécialisateur off-line

Pour une introduction plus complète sur les différences entre spécialisation on-line et off-line vous pouvez vous référer à [12].

2.4 État actuel de la discipline

À ce jour, la spécialisation est une technique qui est encore très peu utilisée. Elle est toujours à ses débuts et si elle s'est avérée utile dans certaines situations, ces cas sont encore assez rares. Les domaines dans lesquels la spécialisation est beaucoup utilisée pour le moment sont principalement ceux de la compilation (plus précisément la phase d'optimisation pendant la compilation) et de la programmation déclarative. C'est dans ces deux domaines que l'on trouve le plus d'applications pratiques de la spécialisation de programmes. En ce qui concerne la compilation, nous avons montré plus haut en quoi l'évaluation partielle lui était utile. Pour ce qui est de la programmation déclarative, on trouve beaucoup d'applications de la spécialisation dans son cas, car spécialiser un programme déclaratif est plus facilement automatisable que de spécialiser un programme impératif, essayons de voir pourquoi.

La différence entre programmation impérative et programmation déclarative est qu'en programmation déclarative, on décrit le quoi, c'est-à-dire le problème. Alors qu'en programmation impérative (par exemple, avec le C ou Java), on décrit le comment, c'est-à-dire la solution. La pratique nous à montré que le problème reste toujours relativement le même, alors qu'une solution peut être définie de nombreuses façons, parfois très différentes. Les programmes impératifs fournissent donc un nombre élevé de fonctions, de procédures, d'opérateurs pour représenter ces solutions, alors que la programmation déclarative nécessite moins de ces éléments syntaxiques. C'est un style de programmation plus épuré, et donc plus facilement cernable, ce qui le rend également plus facile à spécialiser.

Pour le moment, les efforts des chercheurs se sont donc concentrés sur la spécialisation de programme déclaratifs, et plus spécifiquement sur les programmes fonctionnels [3] et logiques [4]. Mais, même si elles sont moins nombreuses, il existe aussi des recherches concernant des méthodes pour spécialiser des programmes impératifs comme le java [14] ou encore le C [1].

Dans la suite de ce mémoire, nous utiliserons le paradigme de la programmation logique pour illustrer les techniques de spécialisations actuelles. Le chapitre suivant est donc consacré à une brève description de ce style de programmation.

3. Programmes qui perçoivent les applications comme un ensemble de fonctions mathématiques.

4. Programmes pour lesquels les composants d'une application sont des relations logiques.

Chapitre 3

Introduction à la programmation logique

La programmation logique est un type de programmation basé sur la logique mathématique de premier ordre. Un programme logique est composé d'un ensemble de règles et de faits définissant des relations entre des objets [4]. Une requête sur cet ensemble suffit pour obtenir de l'information d'un tel programme. Les éléments d'un programme logique sont détaillés dans la suite, on utilisera la notation du langage logique Prolog pour les exemples.

3.1 Les faits

Le fait représente le plus simple type de déclaration en programmation logique, il représente une propriété fixe d'un objet ou une relation entre deux objets. Regardons l'exemple ci-dessous.

```
grand(tony).
felin(tony).
typeAnimal(tony, tigre).
```

La première ligne affirme que tony est grand, la deuxième qu'il est un félin, et la troisième dit que tony est un tigre. Ces trois faits constituent un programme simpliste, grâce auquel on pourra conclure que tony est : grand , un félin et un tigre.

Dans ces déclaration de faits, on peut distinguer ce que l'on appelle les prédicats ('felin', 'grand' et 'typeAnimal'), et les termes('tony' et 'tigre') . Ici, les termes sont plus particulièrement des constantes. On notera que les termes sont utilisés pour représenter les objets manipulés par un programme logique, les prédicats représentent les noms des relations que l'on définit sur/entre ces objets.

3.2 Les requêtes

Si un ensemble de faits suffit à représenter un programme basique, pour retirer de l'information utile de ce programme, il faut utiliser une requête. C'est une déclaration qui demande si une relation entre des objets existe.

```
typeAnimal(tony, tigre)?
```

Il s'agit ici d'une requête simple qui va terminer et réussir. Comme elle réussit, on peut conclure que tony est bien un tigre.

Répondre à une requête consiste à déterminer si cette requête est une conséquence logique du programme, ce qui peut être déterminé par l'application de règles de déductions.

Pour rendre les requêtes réellement intéressantes, nous auront besoin d'introduire les variables.

3.3 Les variables

Tenons compte du fait que nous allons avoir besoin de requêtes plus complexes, par exemple pour savoir à quel type d'animal appartient tony, il serait un peu lourd de devoir utiliser une vingtaine de requêtes du type *typeAnimal(tony, cheval) ?*, *typeAnimal(tony, ornithorynque) ?*,... pour trouver la bonne réponse. C'est pour ça que la programmation logique permet l'utilisation de variables[1]. Pour trouver le type d'animal qu'est tony, nous pouvons, grâce à l'ajout de variables, utiliser la requête :

```
typeAnimal(tony, X)?
```

Sémantiquement cela signifie qu'on demande au système s'il existe un type d'animal pour tony. Où X est la variable (son nom a été choisi de façon aléatoire) qui sera remplacée par le type de tony (s'il existe). Dans ce cas, la requête réussira et donnera comme réponse, *X=tigre*. Cela montre qu'on peut trouver le type de tony (car il est défini dans le programme) et que ce type est *tigre*.

A partir de la notion de variable, on peut introduire celle de *terme* qui est une structure unique de données dans un programme logique [4].

Définition *La définition de* terme *est inductive.*

- *Les constantes et les variables sont des* termes.
- *Un* terme *composé est un foncteur dont tous les arguments sont des* termes. *On caractérise un tel foncteur par son nom et son aritée*[2].
- *Un* terme *composé*[3] *est un* terme.

1. Syntaxiquement, ici il s'agira des mots qui commencent par des majuscules, contrairement aux prédicats et aux constantes qui commencent par une minuscule. Cette notation est empruntée au langage logique Prolog.
2. Nombre d'arguments que prend le foncteur.
3. En anglais *compound term*.

– *Les* termes *qui ne comprennent pas de variables sont appelés* termes *de base*[4].

Voici quelques exemples de termes :

– Termes composés : etudie(arthur,X) , f(1), list(a,list(b,nil)),...
– Termes de base : tony, etudie(arthur,informatique),...

3.4 Les règles

Si on ne s'en tenait qu'aux éléments précédents nous ne pourrions pas écrire de programmes vraiment utiles. Il manque une déclaration importante pour améliorer l'idée de programme logique, il s'agit de la notion de règle.

Une règle permet de définir une relation en fonction d'autres relations, elle est de la forme :

$$A \leftarrow B_1, B_2, ..., B_n \ , \ n \geq 0$$

A est la tête de la règle et les B_i (la partie à droite de la flèche) forment le corps, ce sont des atomes, où plus généralement des litéraux. Les règles, ainsi que les faits et les requêtes sont appelées *clauses de Horn* (ou simplement *clauses*). Pour illustrer le rôle des règles, modifions légèrement le programme décrit plus haut :

```
felin(tony).
grand(tony).

typeAnimal(X, tigre) :- grand(X), felin(X).
```

La dernière ligne est une règle où, la tête est $typeAnimal(X, tigre)$ et les corps est composés de *grand(X)* et *felin(X)*.

Cette règle va permettre de prouver qu'un animal X est un tigre, sans pour autant qu'il soit nécessaire que ce soit encodé directement. Grâce à la règle, on sait maintenant que tous les animaux qui sont grands et qui sont des félins sont des tigres. Comme tony est un grand félin, on peut conclure que c'est un tigre.

Les règles permettent donc de simplifier certaines requêtes et certaines représentations de données, elles permettent de définir de nouvelles relations à partir d'autres relations. Elles peuvent aussi être utilisées pour déduire une implication logique entre différents objets. Cela implique que si on sait que les B_i sont vérifiés, alors A sera également vérifié, et vice-versa.

Nous avons maintenant les structures syntaxiques d'un programme logique, définissons maintenant les actions possibles sur cette syntaxe :

4. En anglais *ground terms*.

3.5 Les substitutions et les unificateurs

Les substitutions et les unificateurs sont nécessaire pour formaliser l'exécution d'un programme logique.

Définition *Une* substitution *est un ensemble fini de paires de la forme X_i/t_i, ou X_i est une variable et t_i est un terme, et où $X_i \neq t_i$. Une* substitution *est souvent notée par une lettre de l'alphabet grec (principalement θ et σ).*

Une substitution pourrait être : $\{Nom/tony, Type/tigre\}$

Appliquer une substitution θ sur une expression E s'écrit $E\theta$, et signifie que dans l'expression E, les variables de E qui apparaissent aussi dans le domaine de θ sont remplacées par le terme correspondant. Par exemple, si on applique la substitution ci-dessus, à l'expression *typeAnimal(Nom,Type)* cela donnera :

typeAnimal(tony,tigre)

Un unificateur est une substitution particulière. L'idée est que pour deux termes t_1 et t_2, on essaye de trouver (quand il en existe) une substitution θ qui *unifie* les deux termes, c'est-à-dire que si on l'applique à chaque terme on aura le même résultat.

Définition *Soit s et t, deux termes. Une substitution θ est un* unificateur *pour s et t si et seulement si $s\theta = t\theta$*

Voici quelques exemples d'unificateurs pour deux termes :

Terme 1	Terme 2	Unificateur
X	a	$\{X/a\}$
$f(X,a)$	Y	$\{Y/f(X,a)\}$
$f(X,g(Y))$	$f(h(Z),Z)$	$\{X/h(g(Y)), Z/g(Y)\}$

Il est possible de *composer* deux substitutions, la manière de procéder est la suivante :

Définition *Soit la* composition $\sigma\theta$ *de $\sigma = \{X_1/t_1, ..., X_m/t_m\}$ par $\theta = \{Y_1/u_1, ..., Y_n/u_n\}$ est obtenue à partir de $\{X_1/t_1\theta, ..., X_m/t_m\theta, Y_1/u_1, ..., Y_n/u_n\}$ en supprimant tous les $X_i/t_i\theta$ tels que $X_i = t_i$ et tous les Y_j/u_j tels que $Y_j \in dom(\sigma)$*

Par exemple, les substitutions $\{X/f(Y)\}$ et $\{Y/g(Z)\}$ auront comme composition : $\{X/f(g(Z)), Y/g(Z)\}$. Ou encore, si on prend $\{X/f(X)\}$ et $\{X/a\}$, leur composition donnera : $\{X/a\}$.

À partir de ces définitions, nous pouvons définir la notion d'unificateur le plus général, ou mgu [5].

Définition *Soit θ un unificateur pour les termes s et t. Il est également un*

5. Acronyme pour le terme anglais *most general unifier*.

most general unifier *pour s et t ssi tout unificateur θ' de s et t est une instance de θ, c'est-à-dire qu'il existe une substitution θ'' telle que $\theta' = \theta\theta''$* .

Par exemple, le mgu de $p(B, y)$ et $p(x, f(x))$ est $\{x/B, y/f(B)\}$ et l'application de ce mgu sur chacun des termes donne $p(B, f(B))$. Dans la suite, quand on cherchera une unification entre deux termes, cela reviendra à dire qu'on cherche leur mgu.

3.6 Résolution et dérivation SLD

La méthode d'exécution la plus utilisée pour les programmes logiques est la résolution SLD [6]. C'est une technique qui permet la transformation d'une requête en une autre requête de la façon suivante :

- sélectionner un atome dans la requête
- unifier cet atome avec la tête d'une règle en utilisant le mgu θ
- remplacer l'atome par le corps de la règle
- appliquer θ sur la nouvelle requête

On appelle *dérivation SLD* une suite de requêtes R et de substitutions θ ($R_1 \rightarrow^{\theta_1} R_2 \rightarrow^{\theta_2} R_3 ... \rightarrow^{\theta_{n-1}} R_n$), tels que chaque R_{i+1} est dérivé de R_i en utilisant θ_i.

La dérivation réussit si la dernière requête est la clause vide ($R_n = \square$). Et elle échoue si aucun des atomes dans G_n ne s'unifie avec la tête d'une clause. Dans la cas où la dérivation réussit, la composition de tous les mgus calculés s'appelle *réponse associée* [7].

3.7 Arbre de résolution SLD

Pour représenter efficacement des résolutions SLD, une manière est de les structurer en arbre (qu'on appellera arbre de résolution SLD). L'idée est que dans un tel arbre, la racine représente la requête initiale, les fils d'un noeud sont les résultats obtenus en sélectionnant un atome et en unifiant l'atome sélectionné avec les têtes des différentes clauses. Chaque branche de l'arbre représente une dérivation SLD. Quand une branche est prouvée (quand elle se termine par la clause vide), on le notera sur l'arbre par un carré ($\square$). Les substitutions effectuées pour passer d'un noeud à l'autre sont notées sur la branche entre les deux noeuds.

Pour construire un tel arbre, il existe plusieurs techniques. Pour les exemples suivants nous nous baserons sur celle qu'utilise le langage Prolog. Le principe est qu'on va d'abord construire un arbre en *profondeur* d'abord en sélectionnant les atomes d'une requête de gauche à droite. Les clauses dont la tête s'unifie avec l'atome sélectionné seront considérées dans l'ordre où elles figurent dans

6. De l'anglais *Selective Linear Definite clause resolution.*
7. En anglais *computed answer.*

le programme. On changera de branche quand on arrivera à une clause vide ou à une impossibilité de continuer, on remontera alors d'un niveau afin d'essayer de prouver une autre branche, s'il y en reste à prouver (*backtracking*). On peut encore ajouter que quand il y a plusieurs éléments à prouver sur un noeud, on sélectionne d'abord celui le plus à gauche. C'est l'élément sélectionné qui sera résolu en premier.

3.8 Comportement d'un programme

Essayons d'illustrer ces principes (résolutions et arbres SLD) au travers de la résolution d'une requête sur un programme complet.

Prenons une variante du programme décrit au début de ce chapitre :

```
//Faits
felin(tony).
felin(garfield).
grand(tony).
petit(garfield).

//Règles
typeAnimal(Nom, chat) :- petit(Nom), felin(Nom).
typeAnimal(Nom, tigre) :- grand(Nom), felin(Nom).
```

Ce programme représente deux types d'animaux au travers de la règle `typeAnimal(Nom, Type)`. En français, ce programme énonce que pour être un chat, il faut être un petit félin, et que pour être un tigre, il faut être un grand félin.

Si l'on veut connaitre le type d'un animal, par exemple celui de *tony*, on peut utiliser la requête suivante :

```
typeAnimal(tony, X)?.
```

Voyons maintenant le comportement de cette requête sur le programme au travers d'un arbre SLD. On retient que pour choisir une clause on prendra toujours la première de la liste en premier lieu [8].

Pour la requête `typeAnimal(tony, X)?`, il y a deux clauses possibles dont la tête s'unifie avec la requête, la première implique la substitution $X/chat$, la seconde $X/tigre$. On va d'abord tester la première, et l'on observe que son corps est composé de deux éléments : `petit(tony), felin(tony)`. On sélectionne le plus à gauche et on essaye de la prouver. Il n'y a aucune tête de clause que l'on peut unifier avec `petit(tony)` dans le programme, la branche va donc échouer. On remonte alors d'un niveau et on teste avec la deuxième clause. Pour que cette dernière soit vérifiée, il faut que `grand(tony)` et `felin(tony)` soit vérifiés. Le

8. Cas particulier pour le langage Prolog, en pratique il existe plusieurs manières de choisir une clause.

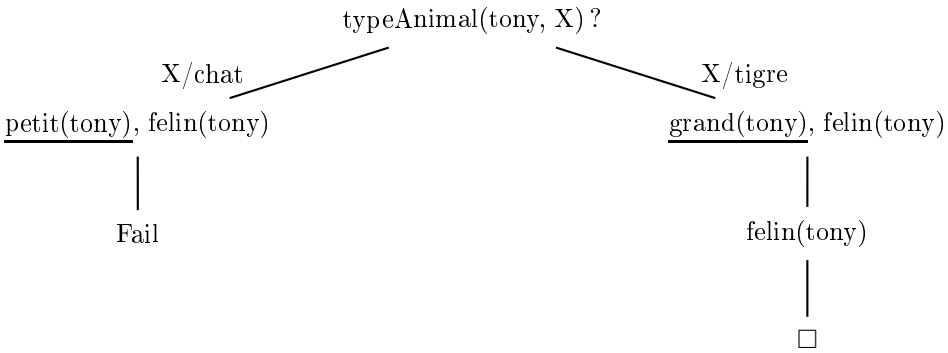

FIGURE 3.1 – Exemple de résolution SLD

premier atome sélectionné, `grand(tony)`, s'unifie avec un fait (règle dont le corps est vide) et donne donc comme résultat la clause vide, on peut donc passer au deuxième atome. Une fois encore, `felin(tony)` s'unifie avec un fait et donne la clause vide. Il n'y a plus rien dans cette branche, elle se termine donc bien. On va alors remonter à nouveau pour voir s'il n'y a plus d'autres branches à résoudre, on voit que non. La requête initiale est finalement bien prouvée et n'a qu'un résultat possible, il s'agit de la composition des mgus calculés dans la branche qui se termine correctement, c'est-à-dire $X/tigre$. Ce résultat implique qu'on peut trouver le type d'animal auquel appartient tony, et qu'il s'agit dans ce cas d'un tigre.

Chapitre 4

La spécialisation de programmes logiques et la déduction partielle

4.1 Principe

Nous en savons assez maintenant pour étudier la spécialisation dans le cadre de programmes logiques. Plus précisément, nous allons travailler avec des programmes logiques *purs*, c'est-à-dire qui ne contiennent pas d'effets de bord, ni de *cuts*[1].

Pour commencer, rappelons qu'en programmation logique, l'exécution d'un programme P sur un atome A revient à construire un arbre SLD pour P avec A comme requête initiale, et pour trouver le résultat, d'en extraire les substitutions calculées sur chaque branche qui réussit [16].

Regardons le programme `append` qui concatène deux listes :

```
append([],L,L).
append([H|X],Y,[H|Z]):- append(X,Y,Z).
```

Si on l'exécute pour la requête `append([a,b],[c],R)`, on obtiendra l'arbre de la figure 4.1. On voit que cet arbre n'a qu'une branche, et qu'elle se termine correctement. En ce rappelant du principe de spécialisation, il est facile d'imaginer qu'un programme spécialisé pour cette requête serait :

```
append'([a,b],[c],[a,b,c]).
```

Ce dernier nous apprend que la concaténation de [a,b] et [c] donnera toujours [a,b,c]. Dans ce cas de figure, la spécialisation n'est vraiment intéressante

1. Fonction particulière au Prolog qui permet d'empêcher le backtracking sur une certaine clause.

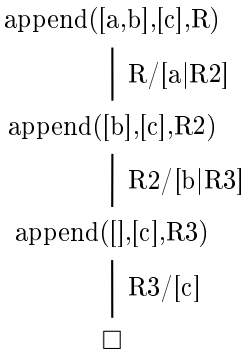

FIGURE 4.1 – Exécution de append sur une requête simple

qu'à condition qu'un appel `append([a,b],[c],R)` apparaisse tel quel dans un programme, ce qui a très peu de chances d'arriver. On préférera en effet coder directement le résultat, plutôt que de faire appel à un prédicat pour un calcul aussi simple.

Par contre, il est plus probable qu'un des deux arguments soit codés quelque part dans le programme en tant que constante. Imaginons par exemple que l'on concatène souvent [c] à une liste quelconque (appelons la X). On pourrait alors spécialiser `append` à partir de la requête moins instanciée `append([c],X,R)` . On dira de ce genre de requête qu'elle est partielle (plus générale) par rapport à la précédente. C'est exactement la spécialisation de ce type de requête qui peut être intéressante pour l'amélioration de programme par la spécialisation. L'arbre en figure 4.2 décrit le comportement de la résolution d'une telle requête, et on constate que cet arbre se termine correctement.

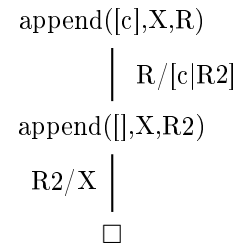

FIGURE 4.2 – Exécution de append sur une requête partielle

À partir de cet arbre on peut trouver le programme spécialisé adapté à la requête de départ $append([c], X, R)$ (nous verrons comment plus tard). Ce programme est :

```
append([c],X,[c|X]).
```

Si on se base sur ce nouveau programme pour résoudre une requête de type $append([c], X, R)$, nous n'aurons plus besoin que d'une étape de dérivation pour

arriver à un résultat. Prenons par exemple la requête $append([c], [a, b], R)$ et regardons les étapes de résolutions de cette requête sur le nouveau programme au travers de l'arbre en figure 4.3. On voit clairement qu'il n'y a plus qu'une seule étape de dérivation, alors qu'il en fallait deux avec l'ancienne version du programme *append*. On a effectivement gagné en efficacité.

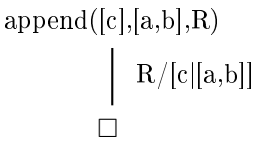

FIGURE 4.3 – Exécution d'une version spécialisée de append

Mais tous les cas d'évaluation d'une requête partielle ne se passent pas si bien. Si l'on essaie plutôt de spécialiser `append(X,[c],R)`, même si le changement dans la requête de départ peut paraitre minime, l'arbre de la figure 4.4 montre que l'on rencontre dans ce cas un problème.

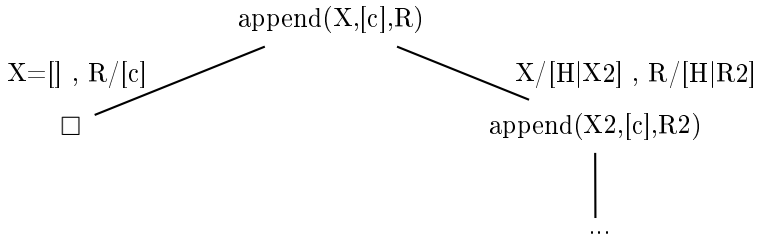

FIGURE 4.4 – Exécution incomplète de append sur une requête partielle

Il apparait tout de suite dans l'arbre que l'exécution de la requête risque de ne pas se terminer, la branche de droite pouvant être dérivée à l'infini. Pour pouvoir spécialiser ce genre de requête, il faut compléter le processus de construction d'arbre SLD par un mécanisme de contrôle de terminaison. On appelle *déduction partielle* l'ensemble de ces techniques.

Pour éviter la construction d'arbres SLD infinis pour des requêtes partielles, on va construire des arbres finis, mais qui peuvent être incomplets. On va donc devoir donner la possibilité de ne pas dériver tous les atomes lors de la construction d'un arbre. Cela conduira à la création de feuilles avec une requête non vide (tel est le cas de `append(X2,[c],R2)}{X=[H|X2],R=[H|R2]`). La spécialisation d'un atome A est donc effectuée à partir d'un *ensemble* d'arbres SLD finis. Les clauses du programme spécialisé sont donc extraites de ces différents arbres, en construisant une clause spécialisée par branche. Pour être sûr que le nouveau programme prendra bien en compte tous les atomes que l'on peut rencontrer lors de l'exécution du programme spécialisé, il faut assurer en plus ce qu'on va appeler la *couverture* [5].

Définition *Soit* $\{A_1, A_2, ..., A_n, \}$ *un ensemble d'arbres SLD finis mais pos-*

siblement incomplets. L'ensemble est couvert si chaque feuille est une instance d'une racine.

Le processus de déduction partielle consiste à partir d'un programme P et d'une requête atomique A, de construire un ensemble couvert d'arbres SLD finis. À partir de cet ensemble, on construira le programme spécialisé comme suit : chaque branche d'un des arbres SLD est transformé en une *résultante*, c'est-à-dire une clause du programme spécialisé.

Définition *Soit un atome A et une dérivation SLD :* $A \rightarrow^{\theta_1} R_1 \rightarrow^{\theta_2} R_2 ... \rightarrow^{\theta_{n-1}} R_n$. *Alors la* résultante *sera :* $A\theta_1\theta_2\theta_{n-1} \leftarrow G_n$.

Les résultantes permettent de trouver un programme spécialisé. Dans l'exemple 4.4, ils sont :

```
append'([],[c],[c]).
append'([H|X2],[c],[H|R2]) :- append'(X2,[c], R2).
```

Ce qui correspond bien à un nouveau programme adapté à la requête, en effet chaque instance de `append([c],X ,R)` pourra être calculée grâce à ce programme.

Si on veut compléter la spécialisation, on peut introduire la de notion filtrage d'arguments [9]. C'est une méthode assez simple qui consiste à retirer les arguments d'un prédicat qui ne sont plus nécessaires. Dans l'exemple ci-dessus, il paraît évident que le deuxième argument n'a plus de raison d'exister, puisque l'on sait que sa valeur sera toujours $[c]$. Notre programme spécialisé donnera finalement :

```
append'([],[c]).
append'([H|X2],[H|R2]) :- append'(X2, R2).
```

Il va de soit que l'argument doit également être supprimé de la requête initiale.

D'un point de vue général, on peut dire que la déduction partielle permet de construire un programme spécialisé à partir d'un ensemble initial d'atomes A, fourni par l'utilisateur. Il faut construire un arbre SLD pour chaque atome présent dans A. Mais il faut également assurer la couverture de ses atomes (ce qui peut nécessiter la révision de l'ensemble de départ), ainsi que la terminaison du processus. Pour contrôler ces deux facteurs, le contrôle est séparé en deux composants :

- Le *contrôle local*, qui contrôle la construction de l'arbre SLD pour chaque atome dans A et qui détermine donc quelles seront les clauses que l'on retient pour les atomes dans A.
- Le *contrôle global*, qui contrôle le contenu de A, et décide quels atomes seront finalement partiellement déduits.

4.2 Contrôle local

Nous allons commencer par appeler règle de déroulement, [2] une fonction qui retourne un arbre SLD partiel à partir d'un programme et d'un atome.

Pour réaliser le contrôle local, il faut trouver une règle de déroulement acceptable. Pour être considérée en tant que telle, la règle doit s'assurer que la spécialisation d'un atome termine et évite l'explosion de l'espace de recherche et la multiplication du travail (répétition des mêmes actions). Le fait qu'une règle de déroulement construise un arbre SLD *fini* est appelée la notion de *terminaison locale*. Plusieurs approches ont été testées dans ce domaine, certaines liées à la profondeur (limiter le déroulement après un nombre fixe de dérivations), d'autres qui choisissent de sélectionner seulement les atomes qui correspondent à la tête d'une seule clause,... Mais ces méthodes ont des failles, elles n'assurent que rarement la terminaison *et* l'efficacité de la spécialisation. Au final, on a découvert que les meilleurs techniques étaient celles basées sur les *ordres bien fondés* ou sur les *ordres presque bien fondés* [3] [15].

Définition *Une relation d'ordre partiel strict $>_s$ sur un ensemble S est une relation binaire anti-reflexive, anti-symétrique et transitive sur $S \times S$. On appelle ordre bien fondé si et seulement s'il n'y a pas de séquence infinie d'éléments $e_1, e_2, ...$ dans S tel que $e_i > e_{i+1}$ pour tout $i \geq 1$.*

Les ordres bien fondés ont été utilisés avec succès pour assurer la terminaison locale dans la déduction partielle [10]. Un exemple souvent utilisé est la relation d'ordre bien fondée de $>$ sur $\mathbb{N}$, en combinaison avec une mesure qui définit la taille d'une requête en fonction du nombre de foncteurs qui y apparaissent. Quand on construit un arbre SLD, on s'assure que chaque noeud a une taille strictement plus petite que celle de son ancêtre direct. Comme il n'existe pas de suite infinie avec chaque élément plus petit que le précédent, on est certain que l'arbre construit sera fini.

Définition *Une relation de presque ordre $\leq_s$ sur un ensemble S est une relation binaire réflexive et transitive sur $S \times S$. On appelle $\leq_s$ un ordre presque bien fondé ssi pour chaque séquence infinie d'éléments $e_1, e_2, ...$ dans S, il y a $i < j$ tel que $e_i \leq_s e_j$*

À nouveau, les relations d'ordre presque bien fondées peuvent être utilisées pour comparer les ancêtres dans une branche d'un arbre SLD en construction. La terminaison locale est assurée si on n'autorise pas la construction de toutes les branches. Une branche ne peut pas être construite si $A \leq_s B$ tel que A et B sont des noeuds de la branche et A est un ancêtre de B. La relation d'ordre presque bien fondé la plus utilisée est l'*enchâssement homéomorphique* que nous appellerons par sa dénomination anglophone, *Homeomorphic Embedding*. Nous développerons ce concept plus en détail dans le chapitre suivant.

2. En anglais *unfolding rule*.
3. En anglais *well-founded orders* et *well-quasi orders*.

4.3 Contrôle global

Dans le contrôle global, il faut assurer la terminaison et la couverture tout en maximisant le degré de spécialisation. L'analyse monovariante [4] résout ces problèmes en gardant au plus un atome pour chaque prédicat dans l'ensemble A des atomes qui seront spécialisés. Quand plusieurs atomes apparaissent avec le même symbole de prédicat, ils sont remplacés par une généralisation (voir exemple plus bas). Cela garantit que chaque prédicat a, au plus, une version spécialisée, ce qui assure la terminaison, car il n'y pas de chaine infinie d'expressions strictement plus générales. Cependant, généraliser des atomes implique une perte de précision dans la spécialisation. Il est alors intéressant d'envisager la polyvariance, la construction de plusieurs versions spécialisées du même prédicat. Le problème est qu'on risque de construire un ensemble infini d'arbres. Il faut alors garantir la terminaison en généralisant certains atomes (racines) ayant le même prédicat. Pour savoir quels atomes généraliser, on peut utiliser les relations d'ordre presque bien fondées comme dans le contrôle local. Cette fois aussi la relation la plus utilisée est l'*enchâssement homéomorphique*($\trianglelefteq$), c'est le sujet du chapitre suivant.

Mais d'abord, expliquons comment fonctionne la généralisation de deux termes.

Définition *Trouver une version plus générale pour deux termes revient à choisir un terme à partir duquel on peut retrouver les deux termes à généraliser. Formellement, cela signifie que G est une généralisation de G_1 et G_2 si et seulement si $\exists\theta_1, \theta_2$ tel que $G\theta_1 = G_1$ et $G\theta_2 = G_2$. Il est évidemment plus avantageux de prendre la* meilleure généralisation*, c'est-à-dire le terme qui est le plus instancié possible (qui évite au maximum les variables).*

Voici quelques exemples de *meilleure généralisation* :

Terme 1	Terme 2	Généralisation
a	b	X
$q(a,c)$	$q(b,Y)$	$q(X,Y)$
$f(a,f(b)$	$f(0,f(q(b)))$	$f(X,f(Y))$

4. On ne construit qu'une seule version spécialisée par prédicat.

Chapitre 5

L'*Homeomorphic Embedding*

5.1 Principe

Comme cité plus haut, une des méthodes les plus utilisées dans le cas du contrôle on-line est l'*Homeomorphic Embedding* [3]. Cette méthode est très efficace pour garantir la terminaison avec un spécialisateur on-line. Son principe est simple, l'HEm est un ordre structurel dans lequel une expression t_1 est plus grande qu'une autre t_2 si elle *enchâsse* cette expression et qu'on note : $t_2 \trianglelefteq t_1$. C'est-à-dire si t_2 peut être obtenue à partir de t_1 en retirant certaines parties de cette dernière.

Définition *La relation d'enchâssement homéomorphique(Hem), écrite* $\trianglelefteq$ *, est définie par les règles suivantes :*

1. $Y \trianglelefteq X$ *pour toutes les variables X,Y.*
2. $s \trianglelefteq f(t_1, ..., t_n)$ *si* $s \trianglelefteq t_i$ *pour un i.*
3. $f(s_1, ..., s_n) \trianglelefteq f(t_1, ..., t_n)$ *si* $s_i \trianglelefteq t_i$ *pour tous les i,* $1 \leq i \leq n$.

On aura par exemple :

- $A \trianglelefteq B$
- $Y \trianglelefteq f(t(0), 1, X)$
- $f(A, B, b) \trianglelefteq f(X, Y, b)$

5.2 HEm dans le contrôle local

Dans le contrôle local, la difficulté réside dans le choix d'une bonne règle de déroulement (voir chapitre 4). Celle-ci doit permettre de dérouler le plus loin possible, mais en garantissant qu'il sera possible de terminer, et cela tant que la spécialisation est jugée profitable. Dès lors, il est intéressant d'avoir un mécanisme qui assure la terminaison en arrêtant le *déroulement* le plus tard possible. L'avantage du contrôle local basé sur HEm, est qu'il n'exige pas une

décroissance stricte des atomes d'une branche d'un arbre SLD, comme c'est le cas pour les ordres bien fondés, il exige seulement une absence de croissance, ce qui lui permet en général d'aller plus loin dans le déroulement.

5.3 HEm dans le contrôle global

L'HEm peut aussi être utile dans le contrôle global, il permet de savoir quand généraliser des atomes. Basiquement, pour chaque nouvel atome A faisant partie des feuilles d'un des arbres SLD construit, le contrôle global vérifie si A enchâsse un des atomes de l'ensemble T_i (qui contient les atomes des racines des arbres partiels déjà construits). Si A n'enchâsse aucun des atomes de T_i alors on peut l'ajouter à cet ensemble, sinon il est généralisé avec l'atome que justement il enchâssait.

Par exemple, si on veut ajouter le terme *p(a,X)* à un ensemble qui contient déjà le terme *p(X,b)*, alors on généralisera les deux atomes et on obtiendra le terme *p(X, Y)*.

5.4 Exemple de spécialisation avec l'HEm

On a constaté au chapitre précédent que certaines tentatives de résolution sur des requêtes partielles menaient à des dérivations infinie. L'HEm est sensé détecter quand la dérivation risque d'être infinie, et exiger alors l'arrêt de cette dérivation. Illustrons son utilisation au travers d'un exemple complet, celui de la dérivation partielle du programme *reverse*, qui inverse le sens d'une liste.

Tout d'abord, notons le programme dans son entièreté, nous aurons également besoin de la définition de *append* pour définir *reverse*.

```
append([],L,L).
append([H|X],Y,[H|Z]):- append(X,Y,Z).

reverse([],R).
reverse([X|Xs],Ys):- reverse(Xs,Zs) ,  append(Zs,[X],Ys).
```

Essayons d'évaluer la requête partielle $reverse(A, R)$. Nous somme conscients que la spécialisation par rapport à une telle requête n'apportera rien, étant donné qu'il s'agit de la requête la plus générale qu'on puisse avoir. Mais nous utiliserons néanmoins cet exemple pour illustrer l'utilité de l'HEm dans un cas de non-terminaison. La figure en 5.1 représente l'arbre de résolution SLD correspondant.

De cet arbre, on voit que l'HEm arrête la dérivation sur la branche de droite, en effet :

$$reverse(A, B) \trianglelefteq reverse(Xs, Zs) \text{ (règle 3) car}$$
$$A \trianglelefteq Xs \text{ et } B \trianglelefteq Zs \text{ (règle 1).}$$

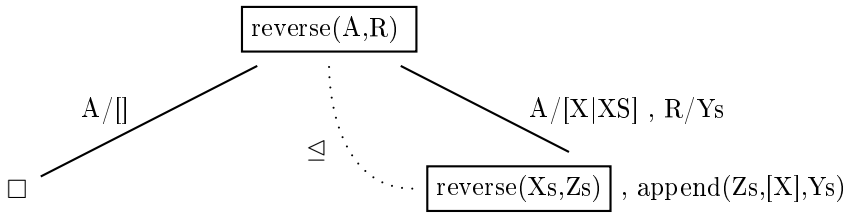

FIGURE 5.1 – Exécution de reverse avec une requête partielle

On doit ensuite passer au contrôle global, où les atomes des feuilles de l'arbre construit seront ajoutés à l'ensemble contenant déjà la racine de l'arbre. Comme $reverse(Xs, Zs,)$ est une instance de $reverse(A, R)$, il ne faut pas ajouter ce dernier. On peut donc déduire après cette phase que les atomes à spécialiser seront : $reverse(A, R)$, $append(Zs, [X], Ys)$.

La spécialisation du premier nécessitera le même arbre qu'en figure 5.1. Par contre, il faut développer un nouvel arbre pour $append(Zs, [X], Ys)$, qu'on peut simplifier en notant $append(A, B, R)$, cela donnera la figure 5.2.

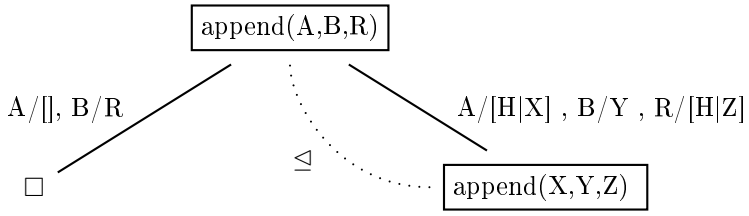

FIGURE 5.2 – Exécution de append avec une requête partielle

Les deux arbres SLD finis représentent le nouveau programme *reverse* spécialisé. Pour l'obtenir, il suffit de prendre les résultantes des deux arbres. Dans ce cas-ci, on retrouvera le programme de départ. Comme on l'avait dit précédemment, spécialiser une telle requête n'améliore en rien le programme de base, mais ne le rend pas moins performant non plus. Cet exemple nous aide à comprendre l'utilité de l'HEm dans un cas de spécialisation. Nous pouvons en conclure que ce mécanisme s'avérera également efficace dans d'autres cas.

Chapitre 6

Spécialisation de méta-interpréteurs

6.1 Principe

Le sujet de ce mémoire s'attarde sur un type de spécialisation un peu particulier : celui de la spécialisation on-line des interpréteurs. Commençons donc par dire qu'un interpréteur est un programme qui lit, traduit, et exécute un autre programme ligne par ligne.

Pour spécialiser un interpréteur, le principe est le même que pour un programme ordinaire. La différence est que le spécialisateur prend comme entrées :
- un programme source.
- des entrées particulières pour le programme, ces entrées étant appliquées à l'interpréteur.

Le programme source est considéré comme l'entrée statique, tandis que l'appel à ce programme est considéré comme dynamique. La spécialisation permet donc de créer un nouvel interpréteur adapté au programme en entrée. Ce nouvel interpréteur est parfois similaire à une version compilée du programme source. Le but est d'obtenir un interpréteur spécialisé qui est au moins aussi performant que l'interpréteur de base (voir figure 6.1).

Pour mesurer facilement les différences de performances entre un interpréteur de base et cet interpréteur spécialisé, il est avantageux que l'interpréteur soit un méta-interpréteur. Pour comprendre ce qu'est un méta-interpréteur, définissons d'abord méta-programme.

Définition *Un méta-programme est un programme qui traite les autres programmes comme des données, des entrées possibles. Ils analysent, transforment et simulent d'autres programmes.*

En programmation logique, écrire un méta-programme est très simple, car

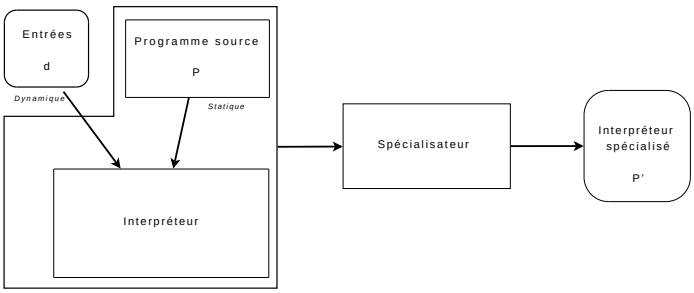

FIGURE 6.1 – Spécialisation d'un interpréteur

les programmes et les données sont syntaxiquement équivalents. Il n'y a pas de distinctions entre programme et données.Quant au méta-interpréteur, ce n'est rien d'autre qu'un interpréteur écrit dans le même langage que le programme qui doit être interprété. Prenons comme exemple le méta-interpréteur Vanilla [4] :

Méta-interpréteur Vanilla

```
solve(true).
solve((A,B)) :- solve(A),solve(B).
solve(X) :- clause(X,Y),solve(Y).
```

C'est un interpréteur très simple. Pour comprendre son fonctionnement, adaptons le programme *append* pour qu'il puisse être exécuté par cet interpréteur. Cela donnera le programme suivant.

Programme `append` adapté pour fonctionner avec l'interpréteur Vanilla :

```
clause(app([],L,L), true).
clause(app([E|Es],Y,[E|R]), app(Es, Y, R)).
```

Cette représentation est avantageuse car les résolutions SLD du programme objet sont immédiates, elles se font en même temps que les unifications et le backtracking sur le méta-interpréteur. Simulons maintenant sont fonctionnement avec la requête $solve(append([a],[b],R))$, le résultat est l'arbre en figure 6.2.

On peut constater que le nombre d'étapes est bien plus important que s'il s'agissait du programme normal, il y aura donc certainement une perte d'efficacité. La réponse finale reste cependant la même, $R = [a, b]$.

À première vue, cet interpréteur n'apporte rien d'intéressant, pourtant Vanilla est un interpréteur très utilisé. Son intérêt principal réside dans la possibilité de construire de nombreuse variantes qui, elles, peuvent apporter quelque chose de plus aux programmes interprétés. L'idée est de permettre l'ajout de fonctionnalités à un programme sans changer son code. Imaginons, par exemple,

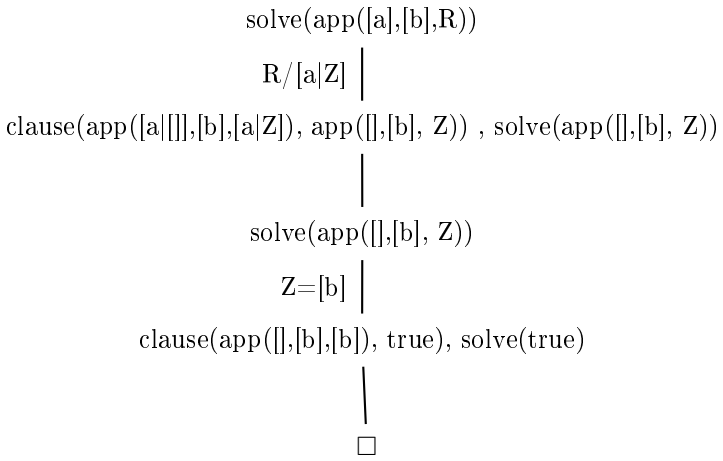

FIGURE 6.2 – Exécution de append sur le méta-interpréteur Vanilla

que l'on aimerait avoir des programmes logiques qui indiquent, à chaque dérivation, la profondeur à laquelle on se trouve dans la résolution. Au lieu de retravailler chaque programme pour qu'il réponde à cette exigence, il suffit de modifier l'interpréteur Vanilla. Le programme suivant est l'interpréteur Vanilla adapté pour indiquer, à chaque dérivation, la profondeur à laquelle on se trouve.

```
solveCompteur(X):-solve(X,0).

solve(true, Depth).
solve((A,B),Depth) :- solve(A,Depth),solve(B,Depth).
solve(X,Depth):-
clause(X,Y), display(Depth),Depth1 is Depth+1 , solve(Y,Depth1).
```

Grâce à cet interpréteur, n'importe quel prédicat déjà décrit pour l'interpréteur Vanilla de base va pouvoir être interprété, en indiquant en plus à quelle profondeur de dérivation on se situe.

Pour ne pas avoir à utiliser l'interpréteur à chaque utilisation de *append*, on peut spécialiser ce méta-interpréteur avec la requête la plus générale $(solve(app(X, Y, Z), Depth))$. Le résultat sera un nouveau programme *append*, pouvant être exécuté seul, et qui donnera la profondeur de dérivation. Le programme que l'on souhaite obtenir, après une spécialisation complète, serait semblable à celui-ci :

```
s_app([],X,Y,D).
s_app([X|Xs,Y,[X|Zs],Depth) :- display(Depth), s_app(X,Y,Z,Depth+1).
```

L'objectif de ce mémoire est de voir dans quelle mesure on peut réaliser cette idée, en employant la déduction partielle on-line. Essayons de spécialiser quelques exemples en utilisant tout d'abord la technique de l'HEm.

6.2 La spécialisation de méta-interpréteurs et l'HEm

L'*Homeomorphic Embedding* peut donc être utilisé efficacement dans la spécialisation d'interpréteur. Illustrons cela en essayant de spécialiser le méta-interpréteur Vanilla avec un programme simple, append (la structure du programme est décrite précédemment).

Pour vérifier si l'HEm est utile dans ce cas précis, tentons d'abord de le spécialiser avec une requête partielle simple, `append(X,[c],R])`. L'arbre de résolution apparaît en figure 6.3.

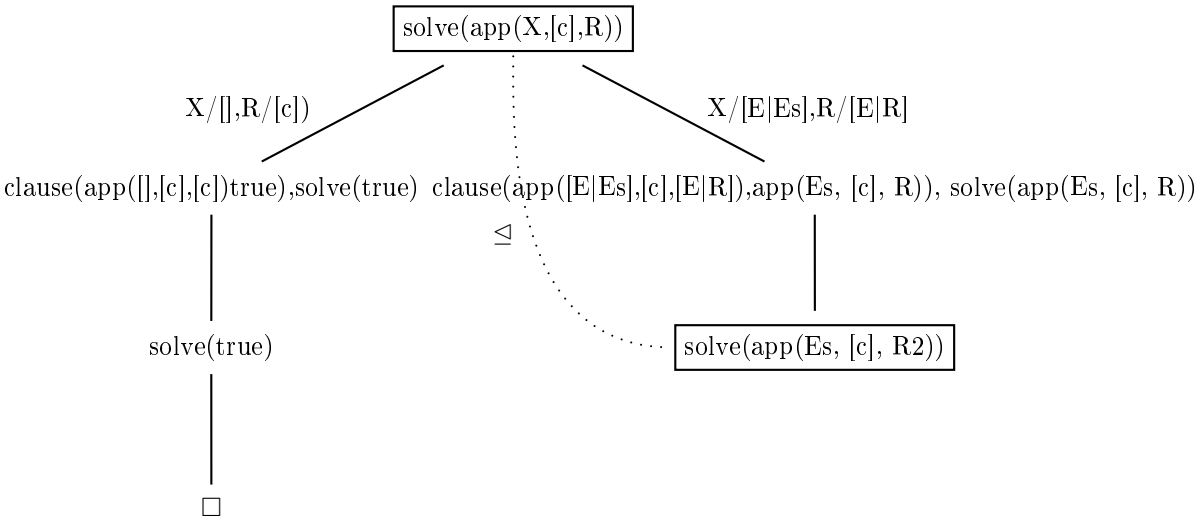

FIGURE 6.3 – Arbre SLD d'une spécialisation de Vanilla sur append

On constate qu'à la fin de la branche de droite, on arrive à une requête très similaire à celle de départ. Le bon sens nous dit qu'il faudrait arrêter la spécialisation ici, car il n'y a plus d'informations à retirer si l'on continue. Mais on peut arriver à la même conclusion grâce au HEm. En effet, il apparaît que $solve(app(X,[c],R)) \trianglelefteq solve(app(Es,[c],R))$, et donc, comme la racine de l'arbre est *enchâssée* par la feuille, on arrête le déroulement. On n'aura donc pas d'autres atomes à spécialiser que $solve(app(X,[c],R))$.

Le programme spécialisé correspond aux résultantes de l'arbre en figure 6.3, cela donnera :

```
solve(app([],[c],[c])).
solve(app([E|Es],[c],[E|R])):- solve(app(Es, [c], R)).
```

Il s'agit bien d'un programme qui permettra de faire tourner toutes les instances de la requête à spécialiser, l'HEm nous a permis de faire une spécialisation efficace.

Cependant il s'agissait ici d'un cas simple. Essayons de spécialiser la requête la plus générale possible pour le même programme que précédemment, c'est-à-dire `append(A,B,R)`. Déduire partiellement cette requête devrait donner comme résultat le programme `append` de base. En quelque sorte cela reviendrait à compiler le programme adapté à l'interpréteur Vanilla pour le rendre indépendant. La figure 6.4 illustre ce cas.

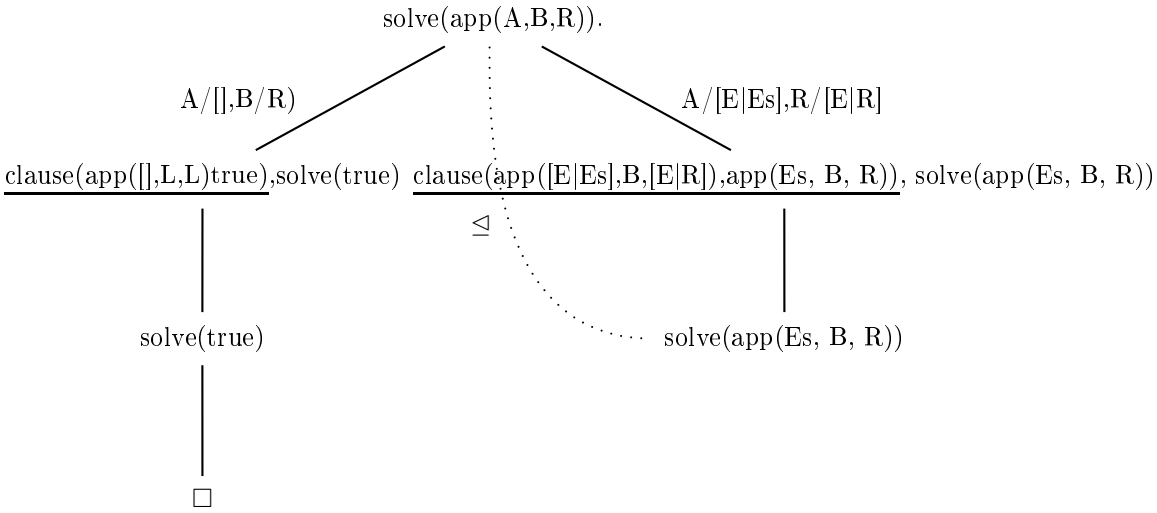

FIGURE 6.4 – Arbre SLD de la spécialisation la plus générale de Vanilla sur append

La spécialisation fonctionne correctement pour ce cas aussi. On peut retrouver, à l'aide des résultantes, le programme suivant :

```
solve(app([],L,L)).
solve(app([E|Es],B,[E|R])):- solve(app(Es, B, R))).
```

Notons que la structure de ce prédicat est identique à la structure du prédicat *append* de base (voir début du chapitre 4) à un foncteur près. La présence de ce foncteur n'est pas problématique, et on peut le retirer en utilisant une technique de filtrage d'argument [9]. On peut donc dire que la spécialisation s'est terminée avec succès.

Cependant, il y a des cas où l'HEm ne sera pas toujours aussi utile. Essayons cette fois de spécialiser Vanilla avec le programme un peu plus complexe `reverse`. Pour écrire `reverse`, on a aussi besoin des clauses de `append`, le programme complet est en fait celui de `append` (plus haut) auquel on doit ajouter les prédicats propres à `reverse` adaptés pour fonctionner avec Vanilla :

```
clause(reverse([],[]), true).
clause(reverse([X,Xs], Ys), (reverse(Xs,Zs), app(Zs,[X],Ys)).
```

La spécialisation de ce nouveau programme donnera l'arbre en figure 6.5.

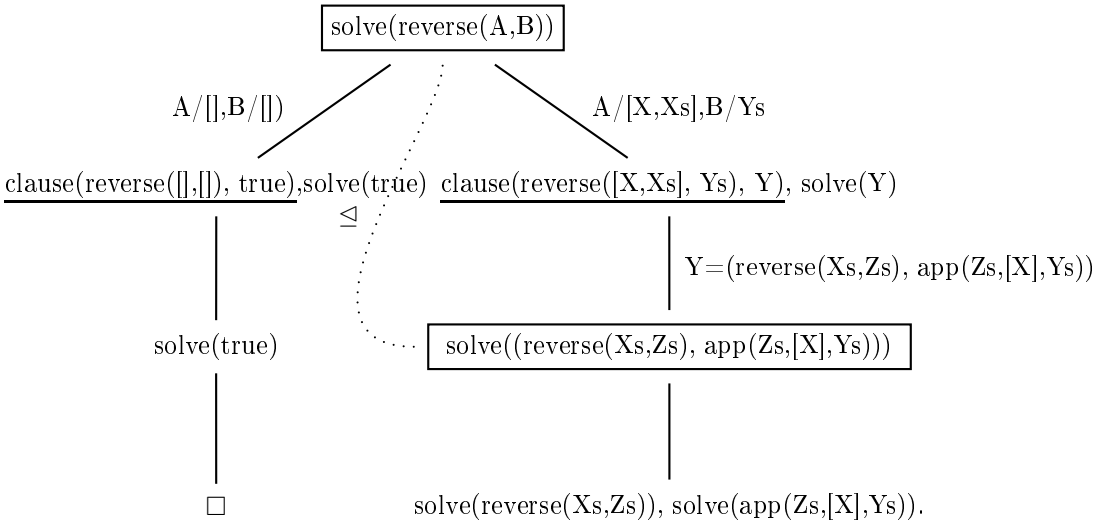

FIGURE 6.5 – Arbre SLD de la spécialisation la plus générale de Vanilla sur reverse

On voit dans cet arbre que l'on est confronté à un cas ou l'HEm exige d'arrêter la spécialisation, alors qu'on pourrait certainement continuer. Le noeud encadré montre où l'arrêt devrait être effectué (*solve((reverse(Xs,Zs), app(Zs,[X],Ys)))*.) car $solve(reverse(A, B)) \trianglelefteq solve((reverse(Xs, Zs), app(Zs, [X], Ys)))$.

Si on continue la spécialisation jusqu'au contrôle global, comme l'un des deux atomes enchâsse l'autre dans l'ensemble des termes à spécialiser, on doit donc les généraliser. Généraliser $solve(reverse(A, B))$ et $solve((reverse(Xs, Zs), app(Zs, [X], Ys)))$ donnera $solve(X)$. Il apparaît alors qu'il n'y a rien d'intéressant à retirer de cette spécialisation, car le prédicat $solve(X)$ a perdu toute l'information statique qui venait de la requête de départ.

Si on avait continué la dérivation de l'arbre jusqu'à un niveau plus loin, on aurait peut-être pu retirer quelque chose d'utile de la déduction partielle. En effet, on voit que si on continue le déroulement, on arrive à deux prédicats plus simples que leur ancêtre $(solve(reverse(Xs, Zs)), app(Zs, [X], Ys))$. Si l'HEm permettait à la spécialisation de continuer, celle-ci aurait peut-être pu être efficace.

Une nouvelle technique va donc être nécessaire pour spécialiser correctement le programme `reverse` pour l'interpréteur Vanilla ; c'est le sujet du chapitre suivant.

Chapitre 7

Le Type-Based Homeomorphic Embedding et la spécialisation de méta-interpréteurs

7.1 Principe

L'article [8] décrit une version améliorée de l'HEm, le *Type-Based Homeomorphic Embedding*,[1] qu'on notera TbHEM. Cette version a été développée car ni l'HEm ni aucune des ses variantes n'étaient suffisant pour résoudre un problème particulier. Ce problème impliquait des pertes d'informations lors de la spécialisation d'un programme utilisant des entiers (voir [8] pour plus de détails). Pour résoudre ce problème, Gómez-Zamalloa , Puebla , Albert et Gallagher ont proposé le *Type-Based Homeomorphic Embedding*, une technique de contrôle de la terminaison qui se différencie de l'HEm par l'ajout d'informations sur le type des termes.

Dans ce chapitre, nous allons introduire cette nouvelle méthode et l'utiliser dans la spécialisation de quelques méta-interpréteurs. Avant de définir le TbHEm commençons par expliquer les notations que nous allons devoir utiliser.

Notation préliminaires

Si P est un programme alors Σ_P est sa signature (peut-être infinie), comprenant les foncteurs et les constantes qui apparaissent dans P. C'est la syntaxe de Mercury [13] qui a été adaptée pour les définition de types. *Type expressions (types)*, les éléments de $\mathcal{T}$, sont construits à partir d'un ensemble infini de va-

1. littéralement *enchâssement homéomorphique basé sur des types*

riables (paramètres) ν_τ et d'un alphabet de symboles Σ_τ (donc ν_τ et Σ_τ sont utilisés pour construire les types dont l'ensemble est représenté par $\mathcal{T}$) ; ces derniers sont disjoints par rapport à un ensemble de variables V et à l'alphabet de foncteurs Σ_P, utilisés pour la construction de termes sur un programme donné P.

Définition *Une règle de type pour un type de symbole $h/n \in \Sigma_\tau$ est de la forme $h(\overline{T}) \rightarrow f_1(\overline{\tau}_1); ...; f_k(\overline{\tau}_k); ...(k \leq 1)$ ou $\overline{T}$ est un n-tuple de différents types de variables, $f_1, ..., f_k, ...$ sont des symboles de fonctions distincts de σ_P, $\overline{\tau}_i (i \leq 1)$ sont des tuples de l'arité correspondantes de $\mathcal{T}$, et les types des variables de l'autre coté, si il y en a, viennent de $\overline{T}$. Une définition de type est un ensemble fini de règles de type où deux règles ne contiennent pas le même type de symbole du coté gauche, et où il y a une règle pour chaque symbole apparaissant dans les règles de type.*

Par exemple, le symbole $list/1$ sera de type : $list(T) \rightarrow []; [T|list(T)]$

Dans la définition ci-dessus, on autorise les règles de type à contenir un nombre infini de cas. Donc, des types infinis comme *entier* sont permis, et ils sont définis par une règle possédant un nombre infini de cas contenant les valeurs numériques.

On considère en plus que chaque partie droite d'une règle est composée de deux composants disjoints, chacun pouvant être vide. Plus précisément, on va structurer une règle de type de cette façon : $h(\overline{T}) \rightarrow F; I$, où l'union $F \cup I$ sont les cas d'une règle. $F \cup I$ n'est pas vide, F est soit vide, soit fini et I est soit vide, soit infini. On dira qu'un type $\tau \in \mathcal{T}$ est dans le composant infini si I n'est pas vide dans la règle qui défini τ. Autrement il sera dans le composant fini. À noter que pour les types du composant infini, il y a une infinité de façon de les séparer en règles ; par exemple $nat \rightarrow F; I$ ou $F = 0$ et $I = \mathbb{N}$, ou $F = \{0, 1, 2\}$ et $I = \mathbb{N} \backslash \{0, 1, 2\}$,...

Définition

Définissons maintenant le *Type-Based Homeomorphic Embedding*, écrit $\trianglelefteq_T$. Pour cela, nous avons besoin d'une relation supplémentaire $\preceq_F$ [2]. C'est une relation sur des symboles de fonctions couplés avec leurs types associés.

Définition *Prenons $\preceq_F$ la relation sur l'ensemble de couple $\Sigma_P \times \mathcal{T}$. $(f_1, \tau_1) \preceq_F (f_2, \tau_2)$ si et seulement si :*

- *Les règles qui définissent τ_i sont de la forme $h_i(Vi) \rightarrow F_i; I_i$, pour $i = 1, 2$*
- *Soit $f_i = f_2$ et $\tau_1 = \tau_2$, soit f_2 est dans le composant I_2 de la règle pour τ_2.*

Par exemple, si $\tau \rightarrow F; I$ avec $F = \{1, 2\}$ et $I = \mathbb{N} \backslash \{1, 2\}$ alors $(1, \tau) \not\preceq_F$

2. Dans la définition originale une autre relation, $\preceq_S$, est également utilisée. Étant donné qu'elle n'était là que par souci d'uniformité avec une autre variante de l'HEm, elle ne sera pas présentée dans ce mémoire.

$(2, \tau)$ et $(1, \tau) \preceq_F (5, \tau)$. Cette relation va intervenir dans la définition formelle du TbHEm qui se trouve ci-dessous.

Définition *Si pour indiquer que le type de X est τ_X on note $X : \tau_X$, alors la relation d'enchâssement sur des termes typés ($\trianglelefteq_T$) est définie par les règles suivantes :*

1. $Y : \tau_Y \trianglelefteq_T X : \tau_X$ *pour toute les variables* X, Y.
2. $s : \tau \trianglelefteq_T f(t_1, ..., t_n) : \tau'$ *si* $s : \tau \trianglelefteq_T t_i : \tau'_i$ *pour un* i *quelconque, quand* $\tau'_1, ..., \tau'_n$ *sont les types respectifs de* $t_1, ..., t_n$.
3. $f(s_1, ..., s_n) : \tau \trianglelefteq_T g(t_1, ..., t_m) : \tau'$ *si*
   - $(f, \tau) \preceq_F (g, \tau')$,
   - $\exists i_1, ..., i_n$ *tel que* $1 \leq i_1 < ... < i_n \leq m$ *et* $\forall j \in \{1, ..., n\}\ s_j : \tau_j \trianglelefteq_T t_{i_j} : \tau'_{i_j}$, *quand* $\tau_1, ..., \tau_n, \tau'_1, ..., \tau'_m$ *sont les types respectifs de* $s_1, ..., s_n, t_1, ..., t_m$.

7.2 Utilisation dans la spécialisation d'interpréteurs

Le but de mon travail dans le cadre de ce mémoire était de vérifier si cette technique pouvait être également profitable dans un autre type d'application, celui de la spécialisation *on-line d'interpréteurs*. Pour arriver à des conclusions utiles, la suite de ce travail va être consacrée au test du TbHEm dans la spécialisation de plusieurs interpréteurs.

Comme vu dans le chapitre précédent, la déduction partielle de Vanilla pour le programme `reverse` posait problème. Reprenons le même exemple, mais en utilisant cette fois-ci le TbHEm au lieu du HEm pour effectuer la spécialisation.

7.2.1 L'interpréteur Vanilla pour append et reverse

Comme on vient de le voir, la particularité du TbHEm est l'attribution d'un type à chaque terme. On va donc prendre le programme `reverse` défini au chapitre précédent et attribuer un type à chacun des ses termes.

Types

Comme c'est le point principal sur lequel ce base la définition de type dans le TbHEm, on va principalement faire la différence entre les types finis et infinis. En effet, au vu de la définition du TbHem, et plus particulièrement de la relation entre fonctions $\preceq_F$, on comprend que ce dont on a réellement besoin pour évaluer un type, c'est de savoir si un foncteur se situe lui même dans la partie infinie ou dans la partie finie de son type. La relation $\trianglelefteq_T$ n'est vérifiée entre deux fonctions que si la deuxième fonction appartient à la catégorie des infinis ou si elle est identique à la première. Prenons un exemple simple, si le type de f est

$\tau_f \rightarrow \{\}\,;\{f\}$ (c'est-à-dire que F=$\{\}$ et I=$\{f\}$). Cela signifiera que $g \preceq_F f$ pour n'importe quel g , peut importe ce qu'il se trouvera d'autre dans les ensembles F et I de τ_f. Mais dans le cas où f ne ferait pas partie de son ensemble infini, alors la condition ne peut être vérifiée, sauf si $g = f$. Pour simplifier la représentation des types, on peut donc faire simplement une distinction entre les foncteurs que nous appellerons *finis* et ceux que nous nommeront *infinis*. Pour l'exemple au-dessus f est considéré comme infini.

Maintenant ne devons trouver un moyen d'attribuer les types aux foncteurs du programme *reverse*. Il n'existe pas à ce jour de techniques efficaces qui permettent de dire si un prédicat doit être considéré comme fini ou comme infini, nous nous baserons donc sur notre intuition pour faire la différence entre foncteurs finis et foncteurs infinis.

Commençons par les plus simples, le foncteur de liste par exemple, '[]'. Ce dernier sera considéré comme infini, étant donné sa définition récursive.

$[] : \tau_{liste} \rightarrow$ infini

Prenons ensuite les différents prédicats propres au programme à interpréter, comme ils sont tous les deux récursifs, on les considéra comme étant infini.

$append(\tau_{liste},\tau_{liste},\tau_{liste}) : \tau_{append} \rightarrow$ infini

$reverse(\tau_{liste},\tau_{liste}) : \tau_{reverse} \rightarrow$ infini

Regardons maintenant le cas des prédicats propres à la structure du méta-interpréteur. Il semble plus difficile de les classer, mais étant donné qu'ils peuvent avoir comme argument une valeur toujours finie, true, considérons les en tant que fini.

$clause(\tau_{clause1},\tau_{clause2}) : \tau_{clause} \rightarrow$ fini
$\tau_{clause1} \rightarrow \emptyset\,;\,\{append, reverse\}$
$\tau_{clause2} \rightarrow \{true\}\,;\,\{append, reverse\}$

$solve(\tau_{solve1}) : \tau_{solve} \rightarrow$ fini
$\tau_{solve1} \rightarrow \{true\}\,;\,\{\tau_{clause1}, (\tau_{clause}, \tau_{clause1})\}$

Il restera finalement le cas de la représentation de la conjonction '(,)', ici, considérons la d'abord comme étant infinie, car elle prend comme argument des types infinis.

$(\tau_{clause1}, \tau_{clause1}) : \tau_{and} \rightarrow$ infini

Maintenant que le programme a été complété par des définitions de type, nous pouvons tester si le TbHEm apporte une solution dans la spécialisation de Vanilla sur *reverse*.

Spécialisation

Puisque nous avons déjà le début de l'arbre de résolution en figure 6.5, nous allons reprendre cet arbre, et tester si en utilisant le TbHEm plutôt que l'HEm,

on doit également s'arrêter au même endroit, en d'autres mots vérifions si :

$$solve(reverse(A, B)) \trianglelefteq_T solve((reverse(Xs, Zs), app(Zs, [X], Ys)))$$

Tout d'abord on voit qu'on a affaire à deux foncteurs, on doit donc vérifier la condition 3 de la définition de TbHEm :

$f(s_1, ..., s_n) : \tau \trianglelefteq_T g(t_1, ..., t_m) : \tau'$

Pour cela, il faut satisfaire deux conditions (voir définition) :

1. $(solve, \tau_{solve}) \preceq_F (solve, \tau_{solve})$ → Oui, car les prédicats sont identiques.
2. $reverse(A, B) \trianglelefteq_T (reverse(Xs, Zs), app(Zs, [X], Ys))$ → Ici aussi il faut que deux conditions soient vérifiées pour se prononcer :
   (a) $(reverse, \tau_{rev}) \preceq_F ((,), \tau_{and})$ → Oui, car on a jugé bon de mettre '(,)' dans la partie infinie.
   (b) $A \trianglelefteq_T (reverse(Xs, Zs)$ et $B \trianglelefteq_T app(Zs, [X], Ys))$ → Ce qui est vrai dans les deux cas.

Ici on à également la condition vérifiée. On doit donc s'arrêter au même endroit de la résolution qu'avec l'HEm. L'utilisation du TbHEm dans ce cas semble ne rien apporter d'utile à la spécialisation. Mais on peut remarquer qu'on aurait pu continuer la dérivation si '(,)' avait été mis dans l'ensemble des objets finis. Or, ce qui nous a poussé à le considérer comme infini était une raison plutôt arbitraire. Donc, de la même manière, on peut considérer que '(,)' pourrait se trouver dans la partie finie. Après tout, même si ces arguments sont infinis, la fonction de conjonction, elle, terminera toujours en une étape. Donc, si l'on considère que '(,)' est de type fini, on peut dire que la relation d'enchâssement n'est pas vérifiée, et dès lors, continuer la dérivation de la branche. C'est ce que l'on fait dans la figure 7.1.

On voit que le changement de type a porté ces fruits dans ce cas précis. En effet, la dérivation a continué une étape plus loin, où on retrouve deux atomes plus simples : $solve(reverse(Xs, Zs))$ et $solve(app(Zs, [X], Ys))$. Le TbHEm exige qu'on s'arrête à cette dérivation, on retrouvera donc ces deux atomes dans l'ensemble des atomes à spécialiser, au même titre que $solve(reverse(A, B))$ (car c'est la racine).

Le contrôle global va bien sûr généraliser $solve(reverse(Xs, Zs))$ et $solve(reverse(A, B))$. Il restera donc deux atomes à spécialiser. La déduction partielle du premier, $solve(reverse(X, Y))$, donne comme on vient de le voir l'arbre en figure 7.1. Il reste donc à créer l'arbre SLD pour $solve(app(Zs, [X], Ys))$, c'est ce qui est fait en dans la figure 7.2.

Après ce contrôle global, on peut déduire une première version du programme spécialisé en prenant les résultantes des deux arbres SLD crées. Nous aurons dans ce cas :

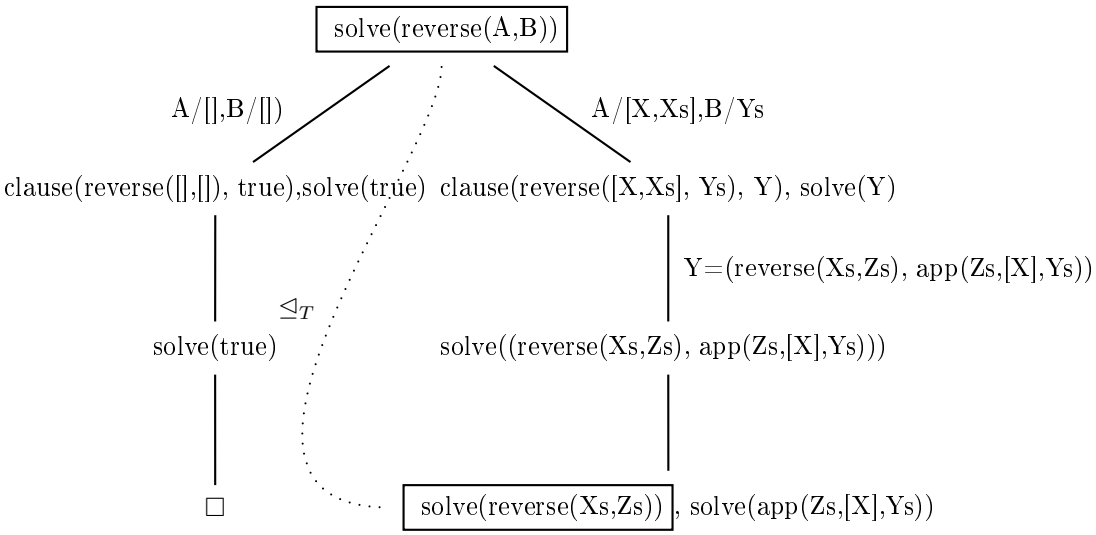

FIGURE 7.1 – Arbre SLD de la spécialisation la plus générale de Vanilla sur reverse avec $\trianglelefteq_T$

```
solve(app([],Z,Z)).
solve(app([E|Es],Y,[E|R])) :- solve(app(Es, Y, R)).

solve(reverse([],[])).
solve(reverse([X,Xs],Ys)) :- solve(reverse(Xs,Zs)), solve(app(Zs,[X],Ys)).
```

On a bien un programme qui peut répondre à toutes les instances de $reverse(X, Y)$. Mais si l'on souhaite encore améliorer ce programme, il reste la possibilité d'ajouter une phase de *filtrage de structure* [9], où on peut rendre le programme plus simple en supprimant certains foncteurs. Par exemple, comme tous les appels à *solve* ont comme argument une valeur dont le foncteur extérieur est *append* ou *reverse*, on peut rendre le programme plus simple en remplaçant ces appels par un appel à un seul prédicat. Ce qui donnera finalement le programme spécialisé suivant :

```
solve_app([],Z,Z).
solve_app([E|Es],Y,[E|R]) :- solve_app(Es, Y, R).

solve_reverse([],[]).
solve_reverse([X,Xs],Ys) :- solve_reverse(Xs,Zs), solve_app(Zs,[X],Ys).
```

Interprétation des résultats

Il est intéressant de voir que la spécialisation d'un programme interprété par Vanilla a abouti à un résultat utile. Mais il serait encore plus intéressant

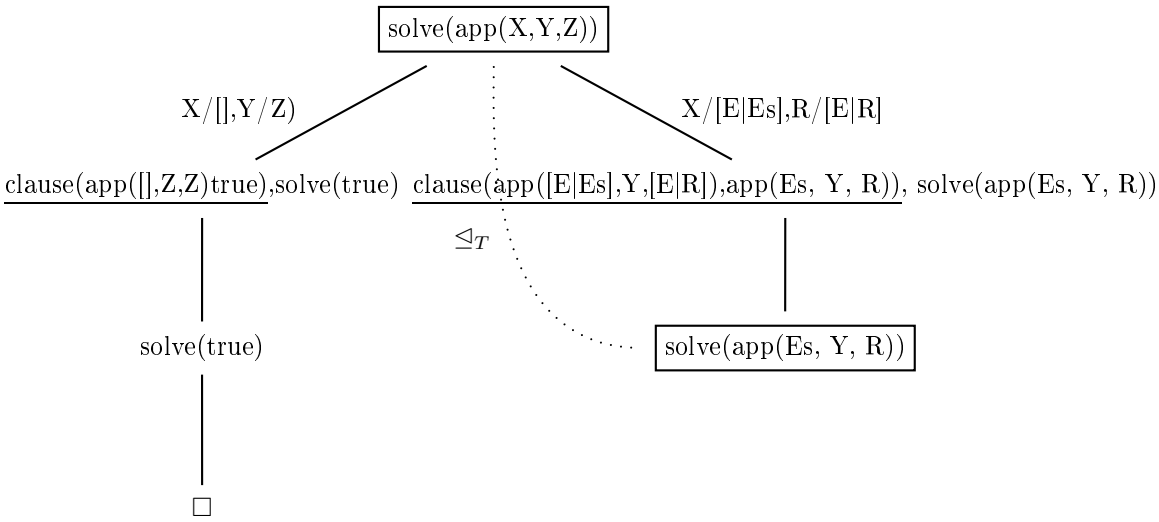

FIGURE 7.2 – Arbre SLD d'une spécialisation de Vanilla sur append avec $\trianglelefteq_T$

de montrer que l'on peut généraliser ce cas, et déduire qu'il est possible de spécialiser efficacement n'importe quel autre programme adapté à Vanilla.

Une condition suffisante pour que la spécialisation soit efficace pour n'importe quel programme est que la terminaison des branches durant la construction de l'arbre SLD dépende uniquement du méta-interpréteur, et non du programme. Comme l'exemple précédent utilise toute les structures du méta-interpréteur et termine correctement, si on arrive à prouver que la spécialisation ne dépend pas du programme à interpréter dans-ce cas-ci, on peut dire que la spécialisation ne dépendra pas du programme interprété dans n'importe quel cas. Essayons donc de voir si les conditions pour la construction d'un arbre complet ne dépendent que du méta-interpréteur.

Tout d'abord, on constate que seul des prédicats identiques peuvent exiger un arrêt de la dérivation d'une branche, cela exclue qu'un *solve* soit enchâssé par un *clause* et que un *solve*(*predicat*1) ne soit enchâssé par un *solve*(*predicat*2), on est donc sûr que l'arrêt de la dérivation ne se fera pas trop tôt.

Ensuite, dans le cas de l'utilisation de (,), il ne faut pas que l'arrêt se fasse trop tôt non plus, comme avec l'HEm. Si on regarde la vérification de la condition, on voit qu'il y a qd même une partie qui dépend des clauses du programmes. C'est la vérification suivante :

1. $(reverse, \tau_{rev}) \preceq_F ((,), \tau_{and})$
2. $A \trianglelefteq_T (reverse(Xs, Zs)$ et $B \trianglelefteq_T app(Zs, [X], Ys))$

.

Alors que la première partie de la condition ne dépend que du type de '(,)', qui est un élément du méta-interpréteur , la deuxième utilise bien des arguments

du programme interprété. Cependant, la deuxième partie n'a été testée que parce que l'on supposait que la première était valide, or on a vu qu'après avoir changé le type de '(,)', la condition $(reverse, \tau_{rev}) \preceq_F ((,), \tau_{and})$ n'est plus correcte. Il sera alors inutile de tester également la deuxième partie.

Finalement, vérifions que l'arrêt ne se fasse pas trop tard. Dans l'exemple précédent, on a stoppé car on a retrouvé la requête de base. Si *append* (ou n'importe quel prédicat) s'était trouvé avant, on aurait du le développer. Mais cela n'aurait pas posé de problème, car l'arbre obtenu aurait été celui de *append* qui se termine correctement également. On aurait finalement réussit à trouver un résultat utile.

Pour que tous les cas de l'interpréteurs Vanilla soient testés, il faut introduire la convention que la conjonction de plus de deux prédicats se fera toujours dans la partie gauche de '(,)'. C'est-à-dire qu'on ne pourra pas avoir $(A, (B, C), D)$ mais qu'on aura à la place $(((A, B), C), D)$. Si on respecte cette convention, on peut conclure que n'importe quel programme écrit pour Vanilla, pourra être spécialisé efficacement.

Maintenant que nous avons démontré que la spécialisation de Vanilla pour *reverse* a réussi et que l'on a montré qu'elle réussira toujours pour n'importe quel programme, il serait intéressant d'essayer d'évaluer partiellement une variante *utile* de Vanilla. Au chapitre 5, nous avions décrit une version de Vanilla comprenant un compteur du niveau de profondeur de dérivation, nous allons maintenant tenter de spécialiser cet interpréteur.

7.2.2 L'interpréteur Vanilla avec compteur de profondeur

Reprenons l'interpréteur Vanilla pour *reverse* et ajoutons lui une fonction qui permet de donner la profondeur du déroulement de la requête, la profondeur sera affichée ($write(Depth)$) après chaque dérivation d'une clause.

Programme

Cela donnera le programme à spécialiser suivant :

```
solveTrace(X):-solve(X,0).

solve(true, Depth).
solve((A,B),Depth) :- solve(A,Depth),solve(B,Depth).
solve(X,Depth):- clause(X,Y), write(Depth), solve(Y, Depth + 1).

clause(app([],L,L),true).
clause(app([H|X],Y,[H|Z]),app(X,Y,Z)).

clause(rev([],R),true).
clause(rev([X|Xs],Ys),(rev(Xs,Zs),app(Zs,[X],Ys))).
```

La première ligne permet de simplifier les requêtes, elle ne sera pas utilisée dans cet exemple.

Types

Comme ici le prédicat *solve* est complété d'une variable entière de profondeur, on doit redéfinir son type. On aura également besoin d'une définition pour le type entier et une pour le foncteur *write*, ce qui donnera :

$entier \rightarrow \{\}\,;\,\{\mathbb{N}\} \rightarrow$ infini

$write(entier) : \tau_{write} \rightarrow$ infini

$solve(\tau_{solve1}, entier) : \tau_{solve} \rightarrow$ infini
$\tau_{solve1} \rightarrow \{true\}\,;\,\{\tau_{clause1}, (\tau_{clause}, \tau_{clause1})\}$

Spécialisation

La spécialisation va se faire sur la requête *solve(reverse(A,B),P)*, où P est une variable de profondeur, le résultat de la dérivation se situe en figure 7.3.

On voit que le développement est presque identique que pour le Vanilla original. On devra également, après le contrôle global, spécialiser les même atomes. On obtiendra donc comme résultat final le programme qui suit :

```
solve(app([],Z,Z),P).
solve(app([E|Es],Y,[E|R]),P) :- solve(app(Es, Y, R), P+1 ).
```

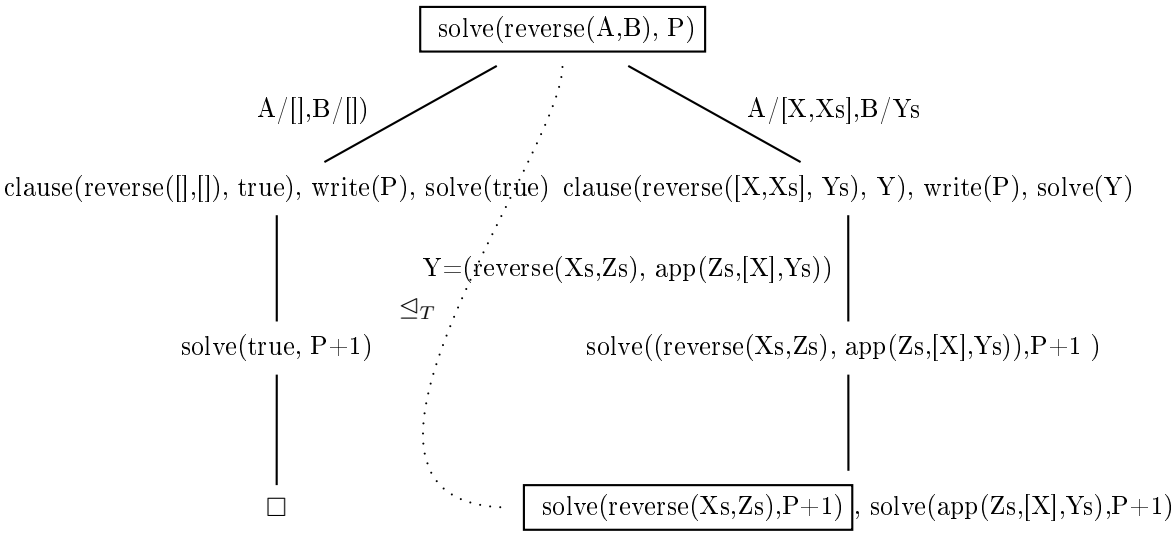

FIGURE 7.3 – Arbre SLD de la spécialisation la plus générale de Vanilla sur reverse avec $\trianglelefteq_T$

```
solve(reverse([],[]),P).
solve(reverse([X,Xs],Ys),P) :- solve(reverse(Xs,Zs),P+1), solve(app(Zs,[X],Ys),P+1).
```

Interprétation des résultats

On constate que la spécialisation de cet interpréteur c'est terminée et semble donner un programme proche de ce qu'on souhaitait obtenir. Cependant, on a perdu lors de la spécialisation le foncteur *write*. Cela signifie que même si on conserve l'information sur la profondeur, celle-ci ne sera plus affichée.

Nous rencontrons donc ici un semi-échec, et on peut en conclure que pour les variantes de Vanilla qui ajoutent des fonctions au Vanilla de base, ces fonctions seront ignorées, car on ne tient compte que des clauses pour la spécialisation. Néanmoins, on a vu que l'on peut ajouter des variables au prédicat *solve* de base, sans que cela pose de problème. De plus, ces variables sont même spécialisées correctement.

De cet exemple, on généralisera en disant qu'une variante de Vanilla pourra être spécialisée correctement, à condition qu'elle n'ajoute aucune fonctionnalité dans le corps de *solve*, et qu'elle se limite à l'ajout de variables dans ses arguments.

Essayons alors de sortir de cette définition pour voir s'il y a moyen de spécialiser des interpréteurs encore plus différents du Vanilla Original.

7.2.3 L'interpréteur Vanilla en liste

Imaginons une autre variante de Vanilla, où l'idée est de stocker dans une liste tous les arguments du prédicat *solve*. L'interpréteur traiterait d'abord le premier élément de la liste, le résultat obtenu serait alors ajouté au début de la liste. Ce système implique que dans beaucoup de cas, la liste sera plus grande après les premiers traitements, ce qui est un problème dans le cas de techniques de style HEm. Voyons si c'est également un problème dans le cas avec le TbHEm.

Programme

L'interpréteur Vanilla en liste est écrit de la façon suivante :

```
solve([]).
solve([A|As]) :- clause(A,B), append(B,As,C), solve(C).
```

Ajoutons à cet interpréteur le programme *reverse* adapté à ce nouvel interpréteur.

```
clause( app([],L,L) , [] ).
clause( app([H|X],Y,[H|Z]), [app(X,Y,Z)] ).

clause( reverse([],R), [] ).
clause( reverse([X|Xs],Ys) , [reverse(Xs,Zs), append(Zs,[X],Ys)] ).
```

Types

Les type n'ont pas beaucoup changés par rapport à l'exemple précédent. Le foncteur de liste ('[]'), ainsi que les différents prédicats propres au programme à interpréter, gardent les même types. Par contre, les prédicats propres à la structure du méta-interpréteur sont légèrement différents, leur type va donc varier.

$clause(\tau_{clause1}, \tau_{liste}) : \tau_{clause} \rightarrow$ fini
$\tau_{clause1} \rightarrow \emptyset\,;\, \{append, reverse\}$

$solve(\tau_{liste}) : \tau_{solve} \rightarrow$ fini

Spécialisation

Essayons maintenant de spécialiser la requête *solve([reverse(X,Y)])*, en construisant l'arbre SLD correspondant (figure 7.4).

Arrêtons nous au cas où il faut tester si *solve([reverse(X,Y)])* $\trianglelefteq_T$ *solve([reverse(Xs,Zs), append(Zs,[X],Ys)])*. Procédons point par point :

1. $(solve, \tau_{solve}) \preceq_F (solve, \tau_{solve}) \rightarrow$ Oui, car les prédicats sont identiques.
2. $[reverse(X,Y)] \trianglelefteq_T [reverse(Xs,Zs), app(Zs,[X],Ys)] \rightarrow$ Ici aussi il faut que deux conditions soient vérifiées pour se prononcer :

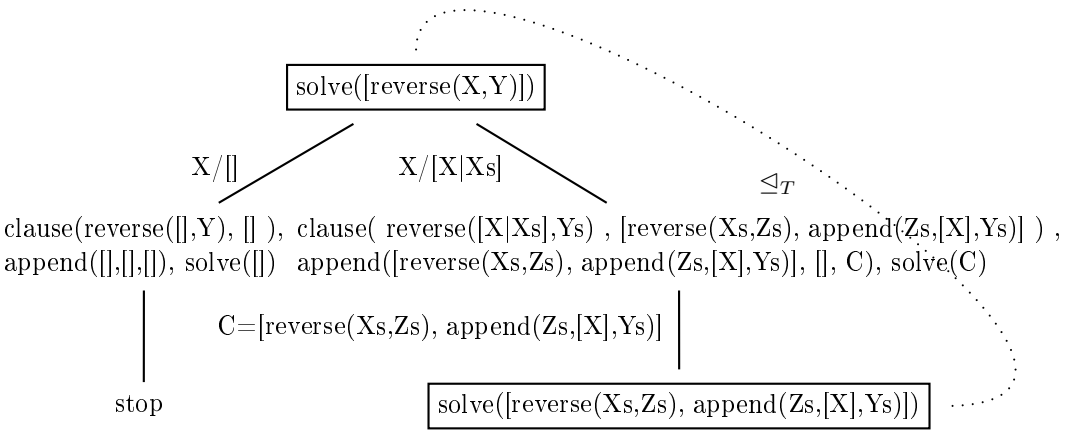

FIGURE 7.4 – Arbre SLD de la spécialisation de reverse sur la variante de Vanilla à base de liste

(a) $([], \tau_{liste}) \preceq_F ([], \tau_{liste}) \rightarrow$ Oui, car les foncteurs sont identiques.

(b) $reverse(X, Y) \trianglelefteq_T reverse(Xs, Zs)) \rightarrow$ Oui (prédicats et variables identiques), cela suffit à prouver la relation d'enchâssement (car le seul élément de la liste de gauche est enchassé par, au moins, un des éléments de la liste de droite).

On doit donc stopper ici le déroulement de l'arbre. Passons ensuite au contrôle global, où on voit que les atomes à spécialiser doivent être généralisés. Dans ca cas, la généralisation de $solve([reverse(X, Y)])$ et de $solve([reverse(Xs, Zs), append(Zs, [X], Ys)])$ donnera le prédicat $solve(X)$. Il est clair qu'aucune spécialisation utile ne pourra en être retirée.

Dans l'exemple précédent, en changeant le type de signe d'un prédicat on a pu continuer à dérouler, et trouver de cette façon des informations pertinentes. Cependant, procéder de la même façon dans cet exemple semble inutile, étant donné que toutes les vérifications sont faites sur des prédicats identiques ([], *reverse*, *solve*), la relation du TbHEm sera donc toujours vérifiée.

Interprétation de résultats

Nous voilà donc confronté à un autre échec de la technique de TbHEm. Nous n'avons donc pas trouvé une solution complète face au problème de spécialisation des interpréteurs. Nous pourrions ensuite continuer de tester le TbHEm avec d'autres exemples, mais on voit déjà ici qu'il est certain que tous les méta-interpréteurs ne pourront pas être spécialisés de façon on-line à l'aide de cette technique dans sa version actuelle. D'autres méthodes vont devoir être décou-

vertes pour, peut-être, arriver un jour à la possibilité de spécialiser n'importe quel méta-interpréteur de façon on-line.

Chapitre 8

Conclusion

Au vu de ce travail, nous pouvons dire que la technique du *Type-Based Homeomorphic Embedding* ne permet pas de résoudre complètement les problèmes rencontrés par la spécialisation on-line d'interpréteurs. Les différents exemples de spécialisation décrits dans ce travail mettent encore en évidence des obstacles que le TbHEm n'a pas pu surmonter. Néanmoins, cette technique a quand même aidé à réaliser des progrès significatifs. En effet, alors qu'auparavant même la spécialisation du méta-interpréteur simpliste Vanilla posait problème, on peut maintenant spécialiser efficacement Vanilla de façon on-line, et ce pour n'importe quel programme adapté à cet interpréteur.

On a également déduit de cette spécialisation qu'un certain type de variantes de Vanilla peuvent également être spécialisées efficacement grâce au TbHEm. Mais dès que la structure du nouvel interpréteur s'éloigne trop de celle du Vanilla original, on peut se retrouver face à des obstacles. Ceux-ci peuvent concerner la terminaison aussi bien que la perte d'information par rapport à l'interpréteur original.

Alors que la spécialisation off-line d'interpréteurs ne pose, quant à elle, plus de problème réel [16], on doit donc malheureusement constater que l'on est encore loin de résoudre tous les problèmes liés à la spécialisation on-line d'interpréteurs, le TbHEm à lui seul n'étant pas une solution suffisante pour trouver une solution à toutes les difficultés rencontrées.

Mais on est droit de supposer que ces problèmes seront un jour résolus, car la piste ouverte par le TbHEm laisse imaginer l'existence d'autres versions du contrôle de la terminaison. Et comme le TbHEm a été développé dans un but différent que celui de la spécialisation des méta-interpréteurs, des nouvelles versions adaptées particulièrement à ce cas spécifique pourraient s'avérées plus efficace, et peut-être résoudre entièrement ce problème de spécialisation on-line d'interpréteurs.

Bibliographie

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[2] L. Augustsson. Partial evaluation in aircraft crew planning. *ACM SIGPLAN Notices*, 32 :127–136, 1997.

[3] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3 :69–116, 1987.

[4] Leon Sterling et Ehud Shapiro. *The Art of Prolog*. The MIT Press, 1986.

[5] Michael Leuschel et Maurice Bruynooghe. Logic program specialisation through partial deduction : Control issues. *Theory and Practice of Logic Programming*, 2 :461–515, 2002.

[6] Singh et McKay. Partial evaluation of hardware. *Lecture Notes in Computer Science*, 1706 :221–230, 1999.

[7] C. Goad. Automatic construction of special purpose programs. *Lecture Notes in Computer Science*, 138 :194–208, 1982.

[8] Miguel Gómez-Zamalloa , Germán Puebla , Elvira Albert et John Gallagher. Typed-based homeomorphic embedding and its applications to online partial evaluation. *Lecture Notes in Computer Science*, 4915 :23–42, 2008.

[9] J. Gallagher et M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9 :305–333, 1991.

[10] Bruynooghe, De Schreye et Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New generation computing*, 11 :47–79, 1992.

[11] Jacobsen, Gomard et Sestoft. Speeding up back propagation by partial evaluation. *Proceedings of the International Conference on Arti cial Neural Nets and Genetic Algorithms*, pages 63–66, 1993.

[12] Jones, Gomard et Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[13] Conway Somogyi, Henderson. The execution of algorithm of mercury : an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29 :17–64, 1996.

[14] Charles Consel et Gilles Muller Ulrik Pagh Schultz, Julia L. Lawall. Towards automatic specialization of java programs. *IRISA*, 1216, 1998.

[15] Wim Vanhoof. *Techniques for on- and off-line specialisation of logic programs.* PhD thesis, K.U.Leuven, 2001.

[16] Michael Leushel, Stephen J. Craig, Maurice Bruynooghe et Wim Vanhoof. Specialising interpreters using offline partial deduction. *Lecture Notes in Computer Science*, 3049 :340–375, 2004.